Unit testing plans

Command Line Utilities

The Command line utilities will be tested in a stand-alone module called cmdline.exe. It will support the command line arguments defined in this document.

Configuration Manager

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest.exe executable should be testable in the DB and the non-DB mode.

Logging Utilities

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Server State Manager

The Server State Manager and the Server Component Framework will be tested independently of specialized components. The routines that require specialization (**StartProcessing, StopProcessing, HandleError** and **UpdateConfig**) will be provided to simply return successfully.

Stress testing plans

Stress testing will require having at least the System Monitor functionality implemented since it is used to drive the server components.

1. Test to repeatedly start, stop, reconfigure the server component.
2. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
3. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
4. Test lost database connectivity
5. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
6. Test error recovery under adverse conditions.
7. Test recovery from running out of memory, thread resources.
8. Test recovery from threads dying.
9. etc.

Coverage testing plans

1. Goal: 100% path flow coverage. Only exceptions for known error conditions that cannot be practically reached (e.g. thread synchronization, etc.)

Cross-component testing plans

The following pair-wise testing will be performed:

1. framework/database (phase 2)
2. framework/messaging (phase 2)
3. framework (System Monitor) /framework (backup Monitor) (phase 3)
4. framework/Web Server (phase 3)
5. framework (System Monitor) /framework (Other Servers) (phase 3)

Upgrading/Supportability/Deployment design

1. Each error condition will be documented with explanations and practical work-arounds
2. Component framework will support enhanced debug option to dump additional debugging information to special log files.

Open Issues

eStream System Monitor Low Level Design

Michael Beckmann

[redacted]

Functionality

The role of the System Monitor is to monitor the state of the Application Servers and SLiM servers within an eStream deployment. In addition, it also manages a back-up System Monitor.

- The System Monitor provides the following key services:
    a. Monitors and reports server load across machines
    b. Monitors server state across machines.
    c. Acts as a communication conduit to the database for configuration information needed by the server components.
    d. Initiates state changes within the server.
        i. Start/Stop servers
        ii. Sends requests for servers to update their configurations
- The system monitor runs as it's own process. Within a multi-system deployment, there will be at least two monitoring processes, each on a different machine:
    a. One monitoring process will act as a primary and the others as backups.
    b. In the event that the primary monitor goes down, one of the backups will take over the primary monitoring responsibilities.
- The monitoring process can re-launch a server side process if a process terminates unexpectedly (this is a configurable option).
- The system monitor manages server state by maintaining regular communication via a heart beat protocol between itself, the backup monitors, and every logical server.
- The monitor will raise an alarm if it does not receive a heart beat response from the servers within a specified period of time.
- The system monitor's heart beat request rate is a dynamically configurable parameter maintained within the database.
    a. The rate can be changed through the administrative interface.
- The system monitor's heart beat supports a light-weight messaging protocol between components to initiate state changes.
    a. Simple request pulse.
    b. Stop request pulse
    c. Configuration request pulse

System Monitor Component Overview:

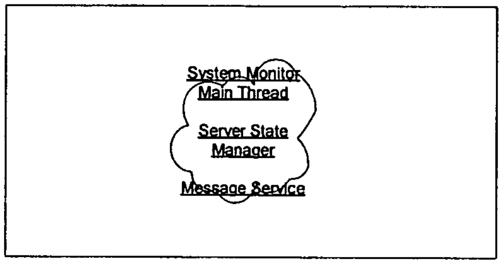
The System Monitor process is extended from *the Server Component Framework* by implementing the plug-able interfaces which includes:

StartProcessing

StopProcessing
UpdateConfig
HandleError

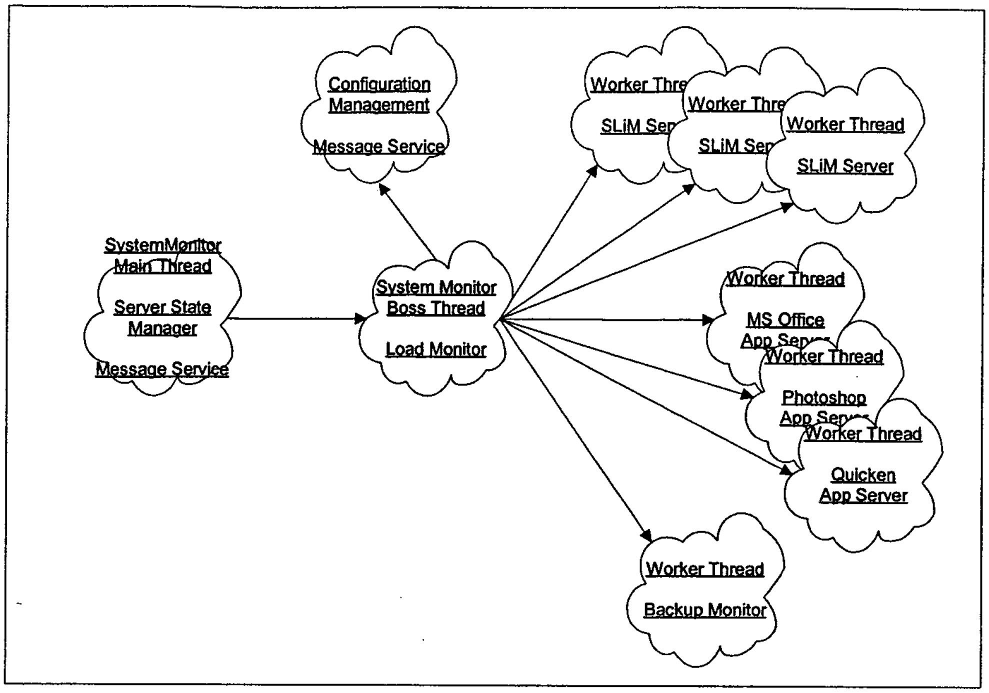
System Monitor States:

When the System Monitor transitions from the STOPPED state to the IDLE state it is maintaining only its main thread which it inherits from the Server Component Framework and runs the Server State Manager. The messaging Service may or may not maintain its own threads. For the purpose of this design we will assume that it is transparent to the System Monitor.



When the System Monitor transitions into the PROCESSING state it employs a "Boss-Worker" parallel programming model:

1. The System Monitor creates a processing thread that acts as the boss thread.
2. The boss thread reads the configuration for all server components and spawns worker threads for each logical server component.
3. Worker threads are allocated, in turn, by the boss to start, monitor, and manage each server process running within a deployment.
4. The boss thread spawns a special thread to handle configuration requests from all the server components.
5. The *Load Monitor* runs as a service within the boss thread.



Synchronization between worker threads and the boss thread is provided through request queues.

Load Monitor:

The *Load Monitor's* role is to aggregate the load information for each server component within the deployment and update aggregated information to the database.

The *Load Monitor* runs in the "boss" processing thread of the *System Monitor*.

For each application the *Load Monitor* maintains a list of servers and their response time. It periodically updates the database with the order lists which are consumed by the SLiM servers. The frequency at which it updates the database is configurable.

The *Load Monitor* maintains three interfaces:

UpdateLoad	Update in memory server sets with the current load information passed in as an argument. This routine is called by the boss thread when the System Monitor is in the PROCESSING state.
DbGetServerSets	Retrieve current server sets in the event that System Monitor crashes and/or starting System Monitor.
DbSetServerSets	Update database with lists of servers for each app

Data type definitions

The System Monitor maintains a few key data structures beyond what is provided by the *Server Component Framework*:

Request Queue:

Each worker thread will maintain its own mutex protected input queue. The input queues are used primarily to communicate requests between threads. A queue entry models, fairly closely, the data contained within the messaging protocol between servers as defined in the *Server Component Framework.* This data can be used for the input queues for each worker thread managing individual work components.

The details of the input queues themselves is defined in the common thread API.

```
struct QueueEntry {
        int Request
        ServerID SenderID
        ServerID RecieverID
        ServerID TargetID
        Data
}
```

Managing Worker Threads:

The System Monitor's boss thread maintains an in memory list of all known servers within a deployment, and their associated worker thread ID. In addition, it maintains information about request queues for each worker thread. This list is maintained as the **SystemMonitorList**. The list is made up of entries which maintains all the necessary information to start/stop/manage/communicate/etc. with a server component.

```
SystemMonitorEntry = struct {
        ServerConfig* ServerInfo;
        Thread ThreadID;
        ThreadQueue* InputQueue;
}

typedef vector <SystemMonitorEntry> SystemMonitorList;
```

Interface definitions

The implementation of the System Monitor is derived from the generic *ServerComponent* class. Refer to the *Server Component Framework* Low Level Design.

Implementing a *SeverComponent* requires the definition of the following methods:

StartServer	**Description:** This routine is called to start a server process and bring it to the IDLE state. It assumes that the server is in the STOPPED state. This routine performs all the necessary initialization and configuration common to all server components. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StopServer	**Description:** The routine StopServer is called to cleanup and terminate a server process. It assumes that the server component is in its IDLE state and is therefore not processing any requests. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StartProcessing	**Description:** This routine initiates all the activities unique to the System Monitor: 1. Launches a boss thread. Call **MonitorServers** as the boss thread entry point. 2. Return immediately Note: The **StartProcessing** routine must return immediately else the Server State Manager will be blocked. **Input:** None. **Output:** Integer value designating return status. **Errors:**

StopProcessing	**<u>Description:</u>** This routine will perform the following activities: 1. Terminates each server component's monitoring thread. This effectively disables the monitor from managing any executing server components including stopping existing components or starting new ones or servicing configuration requests. 2. Disables the load balance manager. Note: The primary system monitor will only execute this interface if a system administrator explicitly changes the monitors state or an error has occurred within the monitor. **<u>Input:</u>** None. **<u>Output:</u>** Integer value designating return status. **<u>Errors:</u>**

UpdateConfig	**<u>Description:</u>** This routine will perform configuration changes specific to system monitor functionality. **<u>Input:</u>** None. **<u>Output:</u>** **<u>Errors:</u>**

MonitorServers	**<u>Description:</u>** This routine is the thread entry point for the boss thread. It is the primary thread for managing and monitoring all the server components. The *Load Manager* runs in the boss thread. 1. Launch the configuration management within its own thread. 2. Construct/initialize *Load Manager* 3. Go to the database and get a list of all the server components. 4. For each server component retrieve its common configurations and create and entry into the **SystemMonitorList** 5. Spawn a worker thread to manage/monitor the backup System Monitor. Call **MonitorServer** as the thread entry point passing in the **SystemMonitorEntry** After the backup monitor has been launched, repeat step 4 for each server component. 6. Loop back and redo steps 2-5 looking for new servers to manage **<u>Input:</u>** **<u>Output:</u>** **<u>Errors:</u>**

MonitorServer	<u>**Description:**</u> Launches and monitors the specified server component process. It can launch server process' locally and on remote machines. It is expected to be the start routine for a new thread. Performs the following steps: 1. Call **StartServerProcess** to launch the server component per the **SystemMonitorEntry**. 2. Open a point-to-point messaging connection to the newly launched server component. 3. Initiate a regular heart-beat request. 4. Enter monitoring loop a. Check for requests from the worker thread input queue b. Process requests by sending request to the server currently being monitored by this thread. c. Listen and wait for response; timeout if wait is too long. If there is a timeout exit thread with an error and let the error handler deal with it. d. If a response is received update state information in **SystemMonitorEntry**. e. Enqueue load information onto Load Balance managers input queue. f. repeat a-f <u>**Input:**</u> **SystemMonitorEntry**. <u>**Output:**</u> Status of the server component on exit. <u>**Errors:**</u> 1. launch error 2. messaging error 3. unexpected server component termination

Primary and Backup System Monitor:

The backup System Monitor needs to be able to take over the primary system monitor responsibilities in the event that the backup loses communication with the primary. The following interface will cause the backup monitor to take over as primary.

SwitchPrimaryMonitor	**Description:** Take over primary monitor responsibilities. 1. Update database validating monitor switch. 2. Shut down the old primary just in case it is still running using by opening a connection to the old primary and sending it a STOP_SERVER request. 3. Go through the same steps as the primary monitor would do when its **StartProcessing** routine is called including starting up a backup system monitor Note: It may be more appropriate to define an error handler that is called when the heart beat is lost. The error handler would go through steps 1 and 2 and then request a state change which would result in the monitor coming on line as a primary monitor. This requires more thought but would be a more elegant solution. **Input:** None **Output:** integer value designating success or failure **Error:**

StartServerProcess	**Description:** This routine spawns the requested process either on the local machine or on a remote machine. 1. Verifies that the target machine is up and running. 2. Launch process specified according to attributes **Input:** Process name, target machine, attributes **Output:** integer return where 0 designates success else error code is returned **Errors:** 1. machine not responding 2. executable not found 3. failure on launch

Component design

Testing design

The System Monitor will provide a command line option to facilitate testing the component in its various testing phases. The option will allow the system to manage its [illegible] through [illegible].

Unit testing plans

The System Monitor can be unit tested in the following phases:

Phase 1: 1. Test state management and messaging capabilities of all server components 2. Test starting, stopping, updating configs	**Dependencies:** 1. Common Server Framework 2. Messaging
Phase 2: 1. Test data persistence and error recovery 2. Test config updates from database 3. Test back-up monitor and switch-over	**Dependencies:** 1. Phase 1 2. Database Connectivity
Phase 3: 1. Test load balance manager and components sending load information	**Dependencies:** 1. Phase 1 & 2 2. Load Balance Manager
Phase 4: 1. Full functionality testing 2. Stress testing 3. Full error recovery testing	**Dependencies:** 1. Phase 1, 2, & 3 2. Web Server 3. App Server/Slim Server

Stress testing plans

Stress testing will be performed at Phase 4 testing. Tests should include:

1. Max # of generic server components on a single machine.
2. Max # of generic components across multiple machines.
3. Max # of generic servers changing state at least once per second
4. Test to repeatedly start, stop, reconfigure each server component.
5. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
6. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
7. Test lost database connectivity
8. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
9. Test error recovery under adverse conditions.
10. Test recovery from running out of memory, thread resources.
11. Test recovery from threads dying.

Coverage testing plans

The System Monitor will achieve 100% code coverage with the exception of error conditions which are possible however difficult to reach in practice.

Cross-component testing plans

The System Monitor interacts with the following components:

1. System Monitor/Database
2. System Monitor/WebServer
3. System Monitor/Server Component Framework (other servers)

Upgrading/Supportability/Deployment design

1. All diagnostics will be documented as to their root cause and workarounds/actions to be taken.
2. The system monitor will support an enhanced debug support which dumps additional information to special debug logs.

Open Issues

1. Need to figure out how to launch a process on a remote machine. May need to use the slave monitor to actually launch the process. But then the question arises … how does one launch the backup monitor?
2. Need to identify a mechanism to ensure that we do not have more than one primary monitor running at any given time.
3. Need to strike a balance on how many threads to spawn. All the server components may be able to be monitored and managed out of a single primary thread which is referred to as the boss thread.

eStream Web Server/Database Low Level Design

Bhaven Avalani

[redacted]

Functionality

The eStream solution provides a set of account, user, and subscription management utilities. These utilities are provided as extensions to the ASP's(Application Service Provider) web server.

There are three categories of users for these utilities: End User, Group Administrator and ASP Administrator. The roles and the capabilities of each of these users are detailed below.

End user for a system is the user who will actually access eStream application using the eStream clients. An end user should be able to:

- Create Account and User attributes. (Username, Password, etc.)
- Change Account and User attributes.
- View all available applications in the eStream system.
- Subscribe/Manage eStream applications.
- View Account Status.
    1. List of applications subscribed.
    2. Status of current subscription.
    3. View/Change Billing information.
    4. View/Change Account information.

A *Group Administrator* is an administrator for a group of users. An individual user is by definition a group administrator for a single user group. Capabilities of a group administrator are:

- (All of single user capabilities).
- Add delete users from a group.
- Manage the active sessions for a group. A group manager should be able to release licenses from active sessions, thereby kicking out active users.
- View the billing information. This will probably need hooks to an external billing system.

An *ASP administrator* manages the overall application system. Capabilities of an ASP administrator are:

- Manage accounts/users/subscription for all users/groups in the system.
- Manage the application data for a subscription system.

    1. Add new applications to the system.
    2. Modify application information for the system.
    3. Provide the pricing mechanism for the applications(?).
- Manage the servers in the system.
    1. Configure a server.
    2. Stop/Start a server. This is accomplished by a message to the Monitor server.
    3. Get load information for a server.
    4. Get logging information for a server.

There are essentially two different types of accounts, which the system will support: Single user account and corporate accounts.

The following licensing mechanisms will be supported by the system.

- Fixed Duration License. (Typically monthly license).
- Fixed Duration Floating License. An example of this is n licenses for k users for a fixed duration.
- Indefinite License.

Description

There are several key issues that need to be determined for the Web Server architecture. The options available in the market to implement these technologies are listed below.

Web Server:

- Apache
- Netscape Server
- Microsoft Internet Information Server

CGI Technology

- Servlet/JSP
    - Tomcat (from Apache group)
    - JRun (from Allaire)
- Active Server Pages (available on NT only)
- NSAPI (C level API available for Netscape and Apache).
- ISAPI (C level API available for IIS and Apache)
- CGI (Perl/C etc.).

Database Connectivity

- JDBC.

- ODBC
- Native.

Database

- SQLServer
- Oracle
- Sybase
- Informix
- LDAP(??)

The overall proposed solution for eStream 1.0 WebServer release is:

Apache + Tomcat(for JSP/Servlet) + JDBC + SQLServer.

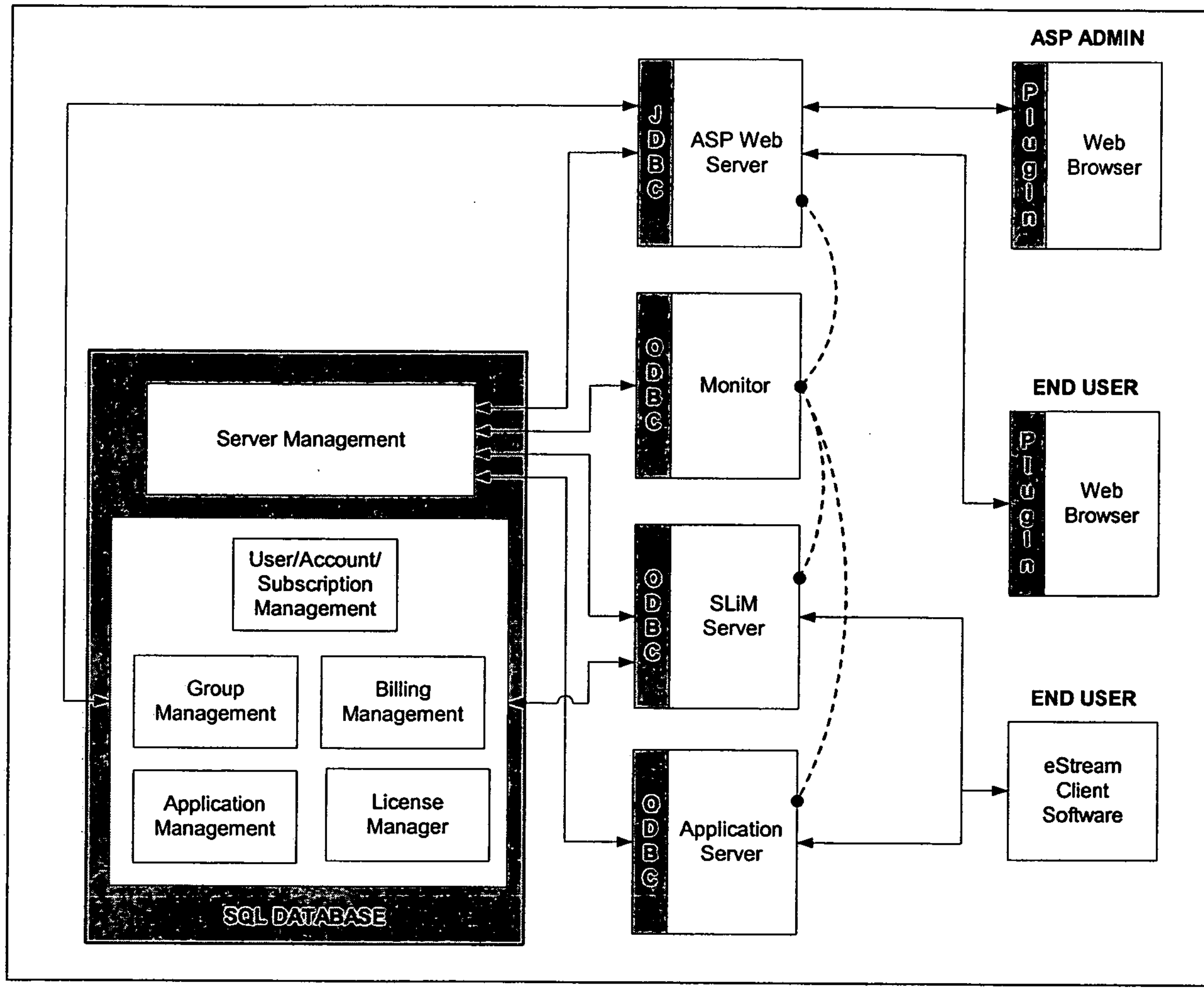
The reasons for choosing this combination for the servers are as follows:

1. JSP/Servlet is the only technology which is available for cross-platform and cross WebServer support.
2. We need to decide on a single web server to develop and test against for release 1.0. Apache is chosen to be the one as it is popular on Unix and NT platforms and it is freely available.
3. Tomcat(Apache group's reference implementation for JSP/Servlet specs) is the preferred CGI technology as it works well with Apache and all other web servers.
4. JDBC is preferred for database connectivity as its database neutral and works well with Java environment of Servlets.
5. SQLServer is the preferred database for release 1.0. This contains the scope for testing and deployment for eStream 1.0.

Since all other servers(App Server, SLM Server and Monitor) are C++ components, the following technology combination will be available for Database Access.

ODBC + SQLServer.

The data model for the eStream 1.0 database essentially consists of two high level components. The database deployment architecture is shown below:



Server Management Component: This component's primary responsibility is to manage the configuration, load and log information for a logical server in the system. The clients to this component are all the servers and administration manager. A detailed list of interfaces for this component is described in the interfaces section.

User/Account/Subscription Management: This component is responsible for maintaining the user account and subscription information for the system. The end user using the end user interface performs the updates to this component. Slim Server will access this component to validate subscriptions. A detailed list of interfaces for this component is described in the interfaces section.

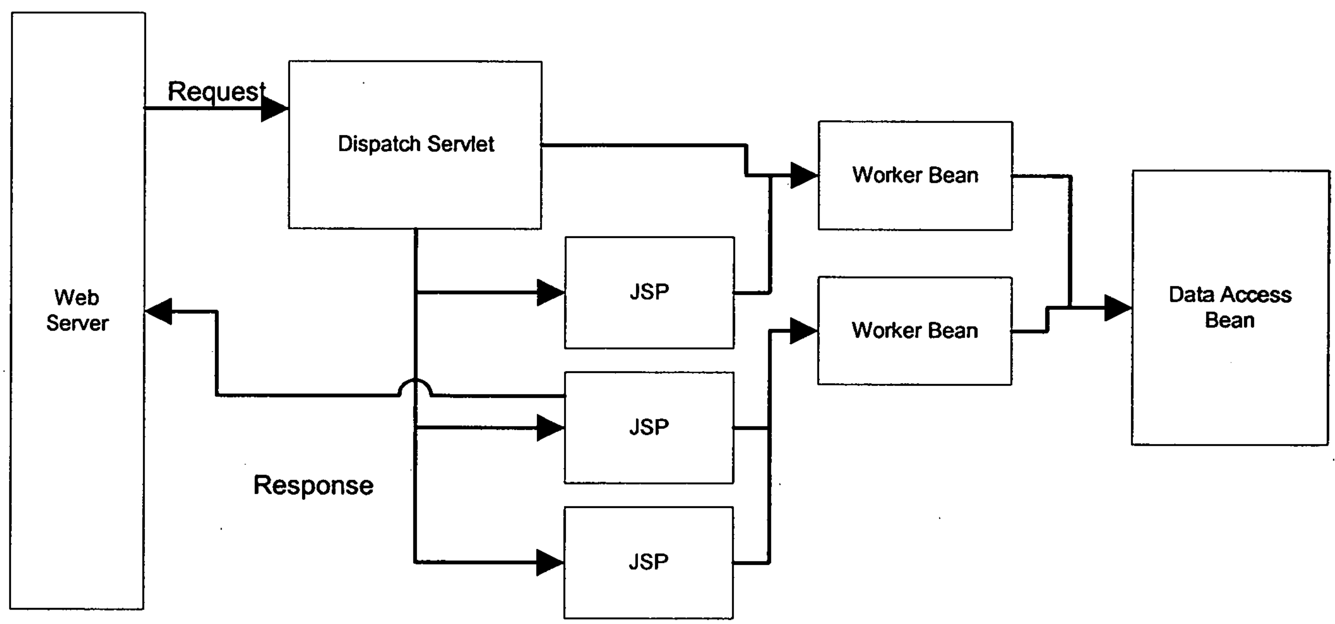
Group Management: This component is useful for managing groups of users. The group administrator can only perform updates to this component. A detailed list of interfaces for this component is described in the interfaces section.

Billing Management: This component's responsibility is to provide interfaces to an external billing system. A detailed list of interfaces for this component is described in the interfaces section.

Application Management: This component's responsibility is to provide application management interface. This component is accessible for updates only by the ASP administrator. A detailed list of interfaces for this component is described in the interfaces section.

License Manager: This component's responsibility is to manage the licenses. SLiM server will check out licenses from the license manager. A detailed list of interfaces for this component is described in the interfaces section.

The architecture for the Web Server extensions implementation is shown below:



The basic elements of this architecture are as follows:

1. Every request into the system goes through a dispatcher servlet. This servlet will perform initialization, initial validation of the request and miscellaneous checks before dispatching the request to a JSP page. A worker bean will responsible for performing the initialization. The processing of the incoming request is performed at this stage. The request is then dispatched to an appropriate JSP page.
2. The JSP page will invoke worker beans to access the dynamic data from the database via the Data Access Bean and the resultant page is sent back to the user.

This architecture is illustrated with the following example.

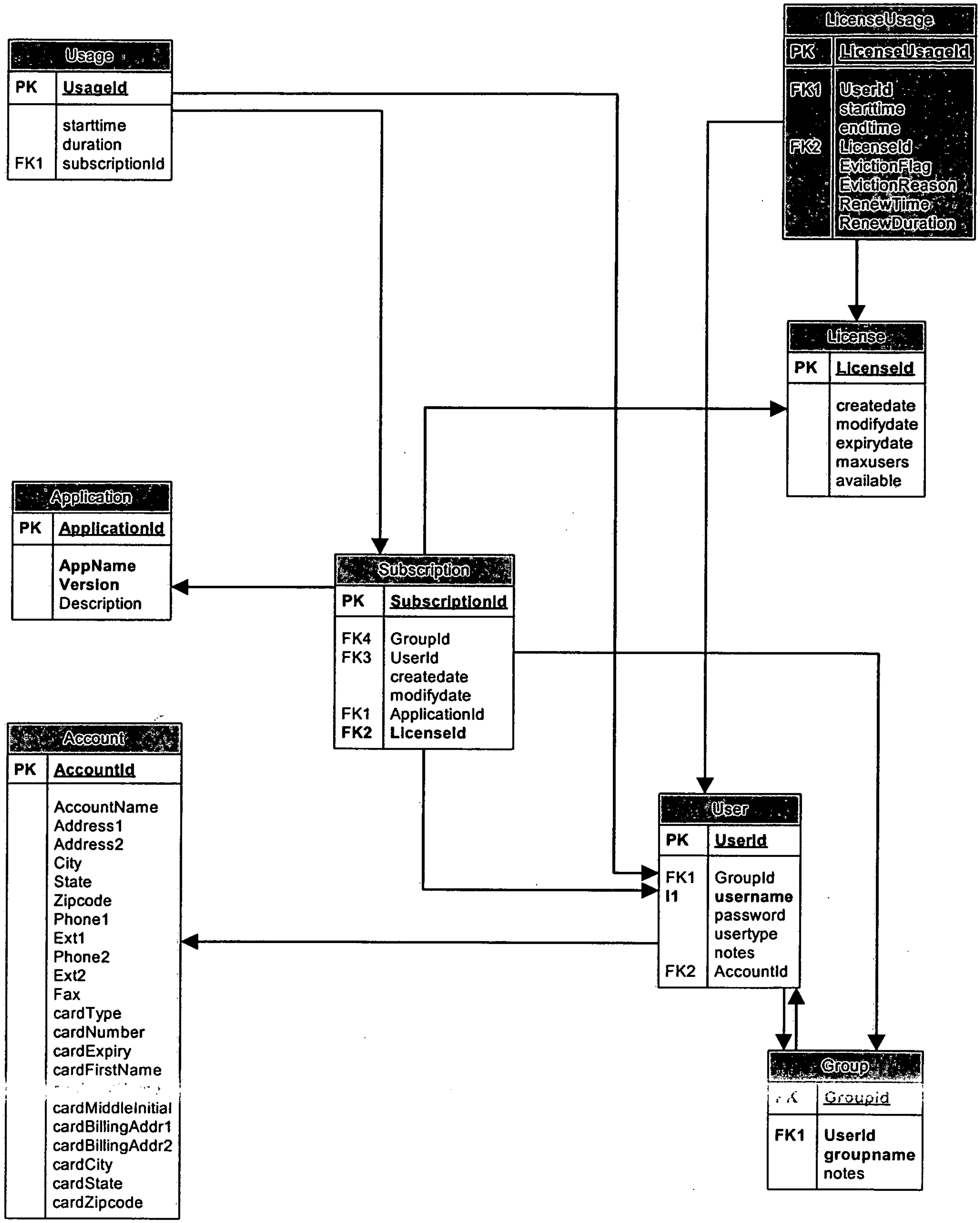
1. User sends in a request to update the username and password information in the database. Inputs are username, old password, new password.
2. The dispatch bean will call the user(worker) bean to:
   a. Validate the user's old password.
      i. The user worker bean will make a request to the data access bean to access the password for the user.
      ii. The two passwords are compared and the result is returned.
   b. If the password was valid then, update the new password.
      i. Call the data access bean to update the password in the database.
   c. Else return failure.
3. Based on the success or failure the dispatcher will dispatch the page request to the appropriate JSP page. (eg. error.jsp on failure and user.jsp on success).
4. The page will invoke the appropriate the worker bean (error bean or user bean) to obtain the dynamic data and send the response back to the user.

The salient features of this architecture are:

1. Presentation and processing logic is separate. Thus, the customer(ASP) can customize the look and the feel of the pages without impacting the processing logic as it is segregated.
2. The data access bean is separated from the worker beans, which are primarily responsible for the business logic. This allows us to change the data access layer (eg enabling LDAP access) in the future without impacting the system drastically.

Data type definitions

The central data structure for Web Server is the database model. The overall database model for user and subscription management is shown below.


Usage
PK UsageId
starttime
duration
FK1 subscriptionId
LicenseUsage
PK LicenseUsageId
FK1 UserId
starttime
endtime
FK2 LicenseId
EvictionFlag
EvictionReason
RenewTime
RenewDuration
License
PK LicenseId
createdate
modifydate
expirydate
maxusers
available
Application
PK ApplicationId
AppName
Version
Description
Subscription
PK SubscriptionId
FK4 GroupId
FK3 UserId
createdate
modifydate
FK1 ApplicationId
FK2 LicenseId
Account
PK AccountId
AccountName
Address1
Address2
City
State
Zipcode
Phone1
Ext1
Phone2
Ext2
Fax
cardType
cardNumber
cardExpiry
cardFirstName
cardMiddleInitial
cardBillingAddr1
cardBillingAddr2
cardCity
cardState
cardZipcode
User
PK UserId
FK1 GroupId
I1 username
password
usertype
notes
FK2 AccountId
Group
Groupid
FK1 UserId
groupname
notes

The important features of this data model are:

Account: Table holding all the billing and contact information for a user or a group.

User: An end user in the system. A user can optionally belong to a group.

Group: A group of users. One of the users in the group is designated as the group administrator. Each group has a unique account associated with it.

Application: This table contains the data about various applications in the supported by the ASP.

License: Each row in this table corresponds to the licensing term for a given subscription. This table also maintains the active count of the licenses checked out.

Subscription: This table contains entries for subscription items. A subscription item consists of user/group, application and license.

Usage: This table contains the runtime information for a system. SLiM server updates this table with access token usage data. A billing system may interface with this table to generate billing data. A reporting system may interface with this table to report on usage patterns.

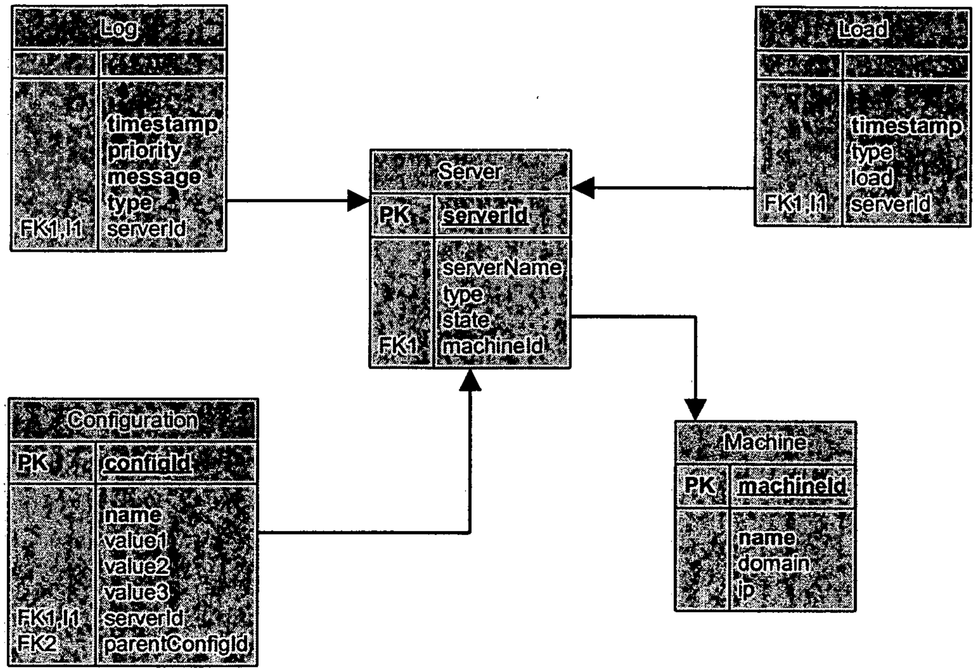
LicenseUsage: This table is responsible for recording checked out licenses in the system.

The data model for storing the server related information is shown below:

PK: Primary Key for the table.

FK: Foreign Key. Used for relations between tables.

I1,I2.. : Index Columns.



The tables in this model are:

Server: This table contains entries for each logical server in the system.

Machine: This table contains entries for each physical server in the system

Configuration: This table contains configuration entries for a given server. The configuration entries can be hierarchical in nature. Each configuration has the following format:

Name Value1 [Value2] [Value3] [ParentConfigId]

Load: This table maintains the historical and real-time load information for a given logical server in the system.

Log: This table maintains the logs for a logical server in the system. The log messages saved here are "major" events in the logical server system. A detailed logs stored in a flat file on the physical machine containing the logical servers.

Global Data Structures:

```
struct ServerTuple
{
int serverId,
int type,
String serverName
};
```

```
struct Couple
{
String name,
String value
};
```

For the Access Token and related data structures, please refer to the SLiM server Low Level Design Document. The interfaces below will discuss some of the API's based on the these data structures.

Interface definitions

The interfaces exposed by various sub-components are detailed below.

Server Management Component:

CreateServer

int CreateServer (ServerConfig* config)

```
Input:
   Server Configuration.
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

UpdateServerConfig

Bool UpdateServerConfig(int serverId, String name, String value)

```
Input:
   Server Id
   Config name and value
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
```

```
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

AddMachine

Bool AddMachine(String name, String domain, String ip)

```
Input:
   Machine name,domain and ip.
Output:
   Success/Failure
Comments:
   Create a physical machine entry.
Errors:
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetServerLog

Bool SetServerLog(int serverId, LogTuple log)

```
Input:
   Server Id
   Log tuple(data structure in the Logging
document.)
Output:
   Success/Failure
Comments:
   Add the log data for a server
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerLog

LogTuple[] GetServerLog(int serverId, int maxrows = 25)

```
Input:
   Server Id
   Maxrows: Maximum number of rows to be re-
turned.
```

Output:
Array of Log tuples(data structure in the Logging document.)
Comments:
Get the log data for a server
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServers

Server[] GetServers()

Input:

Output:
Array of Server tuples(data structure defined above)
Comments:
Get all the server information
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

SetServerState

Bool SetServerState (int serverId, short state)

Input:
ServerId: Unique id for a server
State: State information for a server.
Output:
Bool True/False for success/failure.
Comments:
Update the database with current state information for a specified server
Errors:
INVALID SERVER ID
DB ROW LOCKED
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerState: Obtain the last known state for a specified server

short GetServerState (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   State: State information for a server.
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerConfig: Obtain configuration information for a specified server

ServerConfig* GetServerConfig (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   ServerConfig*: State information for a
server. (ServerConfig data structure is defined
in the server configuration document).
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetLoadData:

void SetLoadData (int serverId, int Load)

```
Input:
   ServerId: Unique id for a server
   Load: Load for the server
```

Output:

Comments:
Monitor may call this interface to persistently store historical load data. It is still not clear if SLM and application servers will store this directly themselves.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerLoad:

void GetServerLoad (int serverId, , int maxrows = 25, int** Load)

Input:
ServerId: Unique id for a server
maxrows: Maximum number of rows to be returned.Default is 25.
Output:
Load: Load for the server
Comments:
Obtain server component load information to manage load balance.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

FlushLoadData:

void FlushLoadData (<tuples> LoadData)

Input:
LoadData tuples containing <server id, server load> values.
Output:

Comments:
Used to flush aggregated load data to the databa

Errors:
```
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

User/Account/Subscription Management Component

CreateUser. This API is used to create user record in the system. Arguments will be Username, Password.

Bool CreateUser(String username, String password)

ValidateUser. This API is used to validate user record in the system. Arguments will be Username, Password.

Bool ValidateUser(String username, String password)

CreateAccount. This API is used to create account records in the system. Arguments will be billing address, credit card information etc.

Bool CreateAccount(String username, <Account Information>couple[])
Input:
Username associated with the account.
An array of names and values for the account.

AddSubscription. This API is used by the end users/group administrators to subscribe to applications.

Bool AddSubscription(<Subscription Information>couple[])
Input: An array of names and values for the subscription.

UpdatePassword. Used to change user information. Password, username etc.

Bool UpdatePassword(String username, String old-password, String new-password);

UpdateAccount. Used to update the account information. Billing Address etc.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

UpdateSubscription. Used to add additional time to a subscription.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

GetUserRecord. Used to get current user configuration.

Couple[] GetUserRecord (String username)

GetAccountRecord. Used to get current account configuration for a user.

Couple[] GetAccountRecord(String username)

GetSubscriptionRecords. Used to get to subscription records in a database. End user may just want to verify what they are subscribed to.

Couple[][] GetSubscriptionRecords(String username)

Output: An array of array of couples containing the subscription information for a given user.

DeleteUser. Used to delete users who are no longer valid in the system. Typically called by the ASP admin.

Bool DeleteUser(String username)

DeleteAccount. Used to delete un-used accounts.

Bool DeleteAccount(int accountId)

DeleteSubscription. Used by the ASP admin to remove subscriptions.

Bool DeleteSubscription(int subscriptionId)

Group Management Component

CreateGroup. This API is responsible for creating group accounts in the database. Called by the group admin user.

Bool CreateGroup(String groupName, String admin, String notes)

AddUserToGroup. Adds a user to a group.

Bool AddUserToGroup(String groupName, String username)

DeleteUserFromGroup. Removes a user from a group.

Bool DeleteUserFromGroup(String groupName, String username)

GetActiveSessions. Gets the active sessions for a group.

Couple[][] GetActiveSessions(String groupName)

Output: An array of array of couples containing the following information for each active session in the system:

Username
LicenseId
StartTime
EndTime
Subscription

Licensing Component

CheckoutLicense: Checks out a license.

int CheckOutLicense(int subscriptionId, long* pStartTime, long* pStopTime)

Inputs:

SubscriptionId: Subscription id of the user.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

RefreshLicense: Refreshes a license.

int RefreshLicense(int LicenseUsageId, long* pStartTime, long* pStopTime)

Inputs:

LicenseUsageId: License usage id.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
EVICTION
```

CheckinLicense: Check in a license

Bool CheckInLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Success/Failure

Comments:

Errors:

```
INVALID SUBSCRIPTION
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

ValidateLicense: Validate that the user has a license checked out.

Bool ValidateLicense(String username, int subscriptionId, int licen-seUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Yes/No.

Comments:

Errors:

```
INVALID USER
INVALID SUBSCRIPTION
INVALID LICENSE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

DBAcquireAccessToken

RPCReturnCodes DBAcquireAccessToken(long SubscriptionId, long* pAccessTokenId, string UserName, string Password, long* pStartTime, long* pStopTime, long* ApplicationId)

IN	SubscriptionId	Id of the subscription being used.
IN/OUT	pAccessTokenId	-1 if this is a first time access.
IN	UserName	Username string.
IN	PassWord	Encrypted Password
OUT	pStartTime	Start time for Access Token validity.
OUT	pStopTime	Stop time for Access Token validity.
IN/OUT	ApplicationId	Id of the application. –1 Default.
OUT	RPCReturnCodes	RPC Return codes.

Processing:

This is fairly complex function. The processing involved in this function call is:

- If this is the first access (ie *pAccessTokenId == -1) then **ValidateUser**
- If the ApplicationId is –1 then **GetAppId**
- If this is the first access (ie *pAccessTokenId == -1) then **CheckoutLicense**
- If this is a renewal request: **RefreshLicense**
- If there is a failure and it is due to eviction: **GetEvictionReason**

Errors:

```
#define    RPCR_USER_AUTH_FAILED
#define    RPCR_ACCESS_TOKEN_INVALID
#define    RPCR_ACCESS_TOKEN_EXPIRED
#define    RPCR_LICENSE_NOT_AVAILABLE
#define    RPCR_LICENSE_ALREADY_HELD
#define    RPCR_EVICTION_NOTICE
```

```
#define     RPCR_EVICTION_MUST_UPGRADE
#define     RPCR_EVICTION_END_MEMBERSHIP
#define     RPCR_EVICTION_NO_PAYMENT
```

DBReleaseAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:
- Update the Usage table with the appropriate information.
- Delete the LicenseUsage record.

Notes:
- We need a mechanism to release un-released access tokens. The way to do this would be to run a stored procedure at demand and at a predefined intervals to do this cleaning up.

EvictAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:
- Evicts an access token.

Billing Component

AddUsageRecord. Called by the SLM server when it releases an access token.

Bool AddUsageRecord(String username, int subscriptionId, date starttime, long duration).

GetUsageRecordsForUser. Used by external billing system.

Couple[][] GetUsageRecordsForUser(String username)

GetUsageRecordsForGroup Used by external billing system.

Couple[][]GetUsageRecordsForGroup (String groupName)

Application Management Component

AddApplication

int AddApplication(String appname, String version, String description)

Inputs:

Appaname: Application name.
Appversion: Application version
Description. Application description.

Outputs:

-1 for failure to add the application.
>0 otherwise. Application ID.

Comments:

Returns an app id for a newly added application.

Errors:

APPLICATION EXISTS
```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetApplicationId

int GetApplicationId(String appname, int version)

Inputs:

Appaname: Application name.
Appversion: Application version

Outputs:

-1 for failure to find the application.
>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetApplicationId

int GetApplicationId(int SubscriptionId)

Inputs:

SubscriptionId

Outputs:

0 for failure to find the application.

>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetSubscribedApplicationIds

Int[]* GetSubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetUnsubscribedApplicationIds

Int[]* GetUnsubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids not subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetApplicationDetail

Couple[] GetApplicationDetail(int appid)

Inputs:

Application Id.

Outputs:

Array of couple for the app id containing:
{appname, appversion, description} values.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

Component design

We will discuss some complex scenarios in this section.

Subscription

Single New User

1. Create the user. **CreateUser.**
    a. If a user already exists, return error message and go back to 1.
2. Create the account for the user. **CreateAccount**
    a. Get the contact information from the user.
    b. Prompt to get the billing information. The user may decide to not give the billing information at this point.

Corporate group admin creating an account.

1. Create the admin user. **CreateUser.**
2. Create the group. **CreateGroup**
3. Create the account information for the group. **CreateAccount.**
    c. Get the contact information from the user.
    d. Prompt to get the billing information.
4. Add users to the group. **AddUserToGroup.**
    a. This method will automatically create the user if they do not already exist in the system.
    b. The list of users is accessible to the Group Admin by querying:
        i. Our database **GetUserRecords** OR
        ii. Some external database. Eg. LDAP directory.

Single User subscribing to an application

1. Validate the user. **ValidateUser**

2. Prompt to get the billing information if the billing information is not already present.
3. Get the list of un-subscribed applications. GetUnsubscribedApplications.
    a. **GetUnsubscribedApplicationIds.**
    b. For each app id returned, get the application details. **GetApplicationDetail**
4. For each additional application user wants to subscribe, call **AddSubscription**

SLiM server checking out an access token to use an application

1. Call **DBAcquireAccessToken**.

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

Stress testing plans

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

Omnishift C/C++ Coding Standard

[redacted]

Omnishift Confidential

The following proposal is based on the C++ coding standards document available at http://www.possibility.com/Cpp/CppCodingStandard.html. This document will concisely present the coding standards from the coding standard document. The reader should refer to the original document (linked above) for a detailed explanation of the standards. This document has the following sections:

NAMES: This section contains the naming schemes.
ERRORS AND ERROR CODES: This section contains the error formats and the error codes to be used by eStream 1.0 client and server components.
FORMATTING: Code layout and formatting guidelines.
COMMENTS: Guidelines for applying comments to the code.
LOOSE END: Loose ends.

Table of Contents:

NAMES

ID	RULE
Variables	• Use upper case as word separators, lowercase for the rest of the word. • No underbars ('_') • First letter of the variable could be upper/lower case. **Examples:** short Status; unsigned long timeOfDay;
Pointers	• Prepend the variable with p. **Examples:** string* pName; char** ppValue;
Class Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • For externally exposed components, use the first 3 words to denote the component. • No underbars ('_') **Examples:** • class ConfigurationManager • class Config
Library Class Names	• Prefix the classname with OT **Examples:** • class OTHttpListener • class OTServer
Class Method Names	Same rule as for class names except for interfaces where the rule is: • Prefix the interface with the component's name. • Method names may optionally start with a lower case letter. **Examples:** • ECMGetFileId • MonitorGetServerSet • monitorInitialize.
Class Attribute Names	• Attribute names should be prepended with the character 'm'. • After the 'm' use the same rules as for class names. • 'm' always precedes other name modifiers like 'p' for pointer. **Examples:** int miLen; char* mpName; string* mpValue;
Global Variables	• Global variables should be prepended with "g". Examples: int gFlag; Logger gLog;

	Logger* gpLog;
Global Constants	• Global constants should be all caps with '_' separators. Examples: const int A_GLOBAL_CONSTANT= 5;
#defines and Macros	• Put #defines and macros in all upper using '_' separators Examples: #define MAX(a,b) blah #define IS_ERR(err) blah
Function Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • No underbars ('_') **Examples:** • int SomeBloodyFunction()

Enum Names	• all upper using '_' separators
	Examples:
	enum PinStateType
	{
	PIN_OFF,
	PIN_ON
	}

File Names	• File should be all lower case • File name format should be <component>_<sub-component>.* **Examples:** monitor_heartbeat.cpp core_configmanager.c

ERRORS AND ERROR CODES

The following pound defines should be used for returning all successes and failures.

```
#define SUCCESSWITHINFO -1
#define SUCCESS 0
#define FAILURE >0 (The number representing an Error ID).
```

All error messages will be prepended with an error code. The format for error code will be as follows:

[ERRORID] [Severity] [Error Message]

where,

1. ERRORID are unique across the system.
2. Severity can be one of the following:
    a. 1-Low : A warning which can be ignored.
    b. 2-Medium: A warning which needs to be looked into.
    c. 3-High: Recoverable error in the component.
    d. 4-Critical: Irrecoverable error. Needs admin assistance.
3. The error message in itself should have the following format:
   [COMPONENT]:[ERROR MESSAGE]:[WORK AROUND]

Error Ids distribution for client and server are as follows:

0-1000 Server Internal Error Codes.
1001 – 8000 Server Error Codes.
8001 – 9000 Client Internal Error Codes.
9001 –16000 Client Error Codes.

FORMATTING

The following formatting policies should be followed by all code.

Braces Policy.

Place the braces in line with the keywords. An example of this is:

```
if ( 0 == a)
{
...
}
else
{
```

```
...
}
```

Indentation/Tabs/Space Policy

Use the standard Visual C++ settings which are(using Tools->Options->Tabs menu):

Indent Size: 4
Auto Indent: Smart
100 previous lines used for context.

White spaces should be spaces and NOT tabs.
VC++: Select "Insert Spaces" option. (This is NOT the default).
Emacs: Refer http://www.delorie.com/gnu/docs/emacs/emacs_205.html

Line Size etc.

1. Line size should not exceed 78 characters.
2. There should be one statement per line. The following piece of code violates this principle.

```
if (a>b) a++;
```

Method/Functions Formats.

1. Methods should preferably be less 50 lines of code.
2. Methods should not have more than 4 levels of nesting.
3. Methods should preferably be re-entrant. Non-reentrant methods should be clearly marked as such.
4. Each method/function should be preceded with a comment describing the method:

```
/****************************************************************
FUNCTION:
INPUTS:
OUTPUTS:
DESCRIPTION:
ERRORS:
****************************************************************/
```

COMMENTS

Every file should start with the following Copyright disclaimer:

```
/* ====================================================================
 * The Omnishift Software License, Version 1.0
 *
 * [redacted]mnishift Technology.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are not permitted
 *
 */
```

1.
2. Every decision should have comments. The following keyword are associated with decisions:
    a. if, else
    b. while, continue
    c. switch, case, default, break
    d. goto
    e. return
3. Every class should have comment header with the following format:

```
/*******************************************************************
CLASS NAME:
DESCRIPTION:
FRIEND CLASSES:
INCLUDES:
LIBRARIES:
*******************************************************************/
```

4. Every function/method should have a header. (described above).
5. Every file should have a header describing the contents of the file.
6. Every directory should have a README describing the contents of the directory.
7. Make GOTCHAS explicit. Use the following format for gotchas.

- **:TODO: topic** *<Author>:<Date>*
  Means there's more to do here, don't forget.
- **:BUG: [bugid] topic** *<Author>:<Date>*
  means there's a Known bug here, explain it and optionally give a bug ID.
- **:KLUDGE:** *<Author>:<Date>*
  When you've done something ugly say so and explain how you would do it differently next time if you had more time.
- **:TRICKY:** *<Author>:<Date>*
  Tells somebody that the following code is very tricky so don't go changing it without thinking.

- **:WARNING:** *<Author>:<Date>*
  Beware of something.
- **:COMPILER:** *<Author>:<Date>*
  Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.
- **:ATTRIBUTE: value** *<Author>:<Date>*
  The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

where an example of the Author:Date format would be: Bhaven Avalani:[redacted]

LOOSE ENDS

The following section notes some loose ends which do not fall in any of the categories above:

1. Always **initialize all variables** every time.
2. Use **header file guards** against multiple inclusions of the header file. The guards would look like:

   ```
   #ifndef ClassName_h
   #define ClassName_h
   ....
   #endif // ClassName_h
   ```
3. Object constructors should just initialize data. (They cannot return errors). Explicit Initialize() calls should be made to do any involved work.
4. Use **continue** and **goto** sparingly.
5. Be **"const" correct**. Use "const" wherever and whenever applicable.
6. All classes must have a **Default Constructor** and a **Copy Constructor**
7. Set the compilation flag "Warnings as error" in Project -> Settings -> C/C++. This will show all warnings as errors.
8. Use std:: namespace for all STL classes.

eStream 1.0 Server Scaling Estimate

*Anne Holler *[redacted] Version 1.0*

Introduction

This document presents an estimate of server scaling for the eStream 1.0 product as compared with its chief competitor, the Citrix product as deployed by Personable. The document presents relevant attributes of the basic application execution model for each of the two products, discusses and gauges the impact of the areas in which server scaling differs between them, considers the effects of additional attributes of the two products on server scaling, & finally summarizes the differences in expected server scaling in terms of a number. Please feel free to challenge the assumptions, methodology, & calculations herein, now & as we move forward through the design & implementation phases.

In the process of developing this server scaling estimate, certain assumptions about user, system, & program behavior are made, & certain design/implementation goals of the eStream 1.0 product are assumed. This material is listed in separate sections at the end of the document for ease of reference.

This work does not intend to imply that the user experience of the eStream 1.0 & Personable/Citrix products is expected to be comparable with respect to the relative server scaling point identified. Interactive response differs between the two products; first-hand experience with server-based applications running on New Moon & Personable/Citrix and client-based applications running on the eStream prototype suggests that the former are sluggish on an ongoing basis with respect to activities such as selecting from pull-down menus & the latter are as responsive as native wrt such activities, with noticeable delays engendered only when heretofore unused portions of the application's functionality are exercised. Reliability also differs between the two products; smooth fail-over of an active Personable/Citrix application to another server is not supported, whereas such fail-over is included in the eStream 1.0 design.

This document does not address an area in addition to server scaling that may be of competitive interest to ASPs; that area is network bandwidth differences between eStream 1.0 & Personable/Citrix. Though it might seem intuitive that sending application pages across a network consumes more bandwidth than sending user input & display output, aggressive client caching ameliorates the traffic associated with application paging, whereas the traffic associated with the display output can be quite substantial, according to Harwood's book "WNT Terminal Server & Citrix Metaframe" [hereafter, HARW99]. It may be worthwhile to collect and compare bandwidth data wrt the two products.

Acknowledgements

Amit Patel provided key insights. Any mistakes/bogosity are mine.

Application Execution Models

The following are basic attributes, with respect to application server scaling, of the application execution models of Personable/Citrix and eStream 1.0:

Personable/Citrix Client/Server Application Execution Model
  Application executes on server
  On app page faults & app data reads, page loaded from server disk
  Server handles incoming network traffic for all keyboard & mouse events
  Server handles outgoing network traffic for bitmap display update

eStream 1.0 Client/Server Application Execution Model
  Application executes on client
  On app page faults & app data reads, page loaded from client cache, excpt miss to server
  Server handles incoming network traffic for client cache misses
  Server handles outgoing network traffic for client cache misses

Application Server Scaling Comparison

The impact on server scaling of executing applications on the client, rather than on the server, is expected to be large. Processor & main memory overhead are associated with executing applications, including the overhead for fielding interrupts such as mouse & keyboard events. According to HARW99, processing power for running applications is typically the biggest Citrix product bottleneck, and that is reinforced by the variance in Citrix scaling numbers reported wrt Personable [Ernie] and wrt another ASP [Amit], for which the main differences seem to be application execution overhead. HARW99 indicates that Citrix scales at 10 to 45 applications per processor, largely due to execution overhead (though some overhead is due to processing page faults & accessing application data, which is considered in the next paragraph). It is somewhat difficult to understand how to model the relative benefit for this difference (in the sense that an *infinite* number of applications can run on a processor that they are not actually using to execute!); it seems conservative to assume we get at least as much benefit from this factor as we do from reducing cache miss overhead on the server, so let us have this factor double whatever benefit we project from the factor considered in the next paragraph.

The impact on server scaling of processing application page faults & application data reads on the client in most cases, rather than on the server, is expected to be measurable. For the purposes of the server scaling estimate presented in this document, let us assume that the server overhead to fetch a page from disk, whether the request came from server execution or from client request, is comparable. (Although we can construct file server technology in which client requests take less time than server requests, let us assume that the overhead to encrypt the response consumes that time savings). Also, let us assume that the same number of page faults occur on the client & on the server (which server partitioning for Personable/Citrix could render untrue.) Hence, the difference in server

overhead for application file accesses is estimated to be equivalent in scale to the reduction in the number of references, which is derived from the client cache miss rate. Assuming a client cache miss rate of 2%, each eStream application server can handle 50 times as many clients with respect to this attribute of server overhead. Doubling that amount due to the factor described in the previous paragraph, we estimate that each eStream application server can handle 100 times as many clients as each Personable/Citrix application server.

It is somewhat difficult to know how to compare the network interface overhead component of server scaling between eStream 1.0 and Personable/Citrix. The loading constraints associated with executing applications on the server are expected to limit the amount of network overhead presented to a Personable/Citrix application server. Depending on the kind of application, significant traffic is generated to support client displays, but HARW99 identifies network bandwidth overhead [discussed in the Introduction section] – not server overhead - as the scaling problem engendered by this traffic. Each eStream 1.0 client generates little server network overhead, but the reduction in server load due to client application execution allows more clients to be connected to a given server, possibly straining the network-oriented portions of an eStream application server. At this point, let us assume that server network interface overhead does not materially impact the relative server scaling of the two products.

SSL encryption can dramatically decrease server scaling; by more than 70%, according to Igor Balabine. He indicates that third party SSL accelerators should be employed to remove this overhead.

Additional Server Scaling Considerations

For eStream, application installation induces load on the application server to deliver to the client the contents of the AppInstallBlock, which may contain registry & file spoofing information, initial cache & profile data, file system structure information, etc. Personable/Citrix does not have a comparable feature. Installation is expected to be an infrequent occurrence and Omnishift is expected to suggest/provide mechanisms to smooth out (or specially handle) high peak demand at particular points in time, including product or application launch/upgrade. Let us assume that this (managed) overhead reduces application server scaling by the equivalent of 0.5% cache miss. Adjusting the running total by this amount implies that an eStream application server can handle 67 times as many clients as a Personable/Citrix application server can.

For eStream, client prefetching from the server can cause application server overhead that does not have a counterpart on Personable/Citrix [except for page fault clustering ;-) !]. Given that eStream 1.0 is targeted at fat clients, client prefetching (which is redundant wrt a comparably warmed client cache) is expected to be used sparingly. Let us assume that prefetching adds the equivalent of an additional 0.5% cache miss rate; the intuition here is that prefetching is essentially engendered when cache misses occur (i.e., when we are exercising parts of the app we have not exercised before) and that we get unneeded pages a measurable percentage of the time. Updating our running total by this amount

means that an eStream application server can handle 50 times as many clients as a Personable/Citrix application server can.

Both eStream & Personable/Citrix products include several logical servers in addition to the application server. Both have an ASP Web Server portal to an ASP's account services, from which a user can obtain billing information, get a list of available applications, subscribe to new applications, etc. For both, the ASP web server interfaces in some way with an account database of presumably comparable complexity. Both eStream & Personable/Citrix have server functionality involving getting the license to run an application, which is expected to cause similar database overhead; in eStream 1.0, this process involves getting an AccessToken from an ADRM server. However, Personable/Citrix does not have eStream 1.0's concept of renewing an AccessToken. It is expected that the eStream 1.0 design will take special care that renewal does not add significant overhead to the eStream 1.0 ADRM server (by specifying nontrivial billing granularity & AccessToken renewal frequency, by ensuring renewal is lightweight, perhaps by having some explicit mechanism aside from AccessToken renewal for token cancellation in the event of client failure/disconnect, etc.). eStream 1.0 has the concept of records containing application profile information being uploaded to a server; it is not known that Personable/Citrix has any comparable feature (although the product likely has much other user information recorded at the server, including preferences, etc). It is expected that eStream 1.0 design will emphasize minimal server impact for handling this data. [Profile data may not be a core deliverable for competing with Personable/Citrix.] In summary, it is assumed that server scaling for auxiliary servers is comparable between eStream 1.0 & Personable/Citrix.

Conclusion

Based on the discussions in the previous sections, eStream Server Scaling expected to be significantly higher than that of Personable/Citrix. Considering client execution benefits, client caching benefits, application installation overhead, and prefetching overhead, eStream server scaling is expected to be approximately 67 times higher than the Personable/Citrix competitive product. For some given Personable/Citrix application server that can handle 20 clients, an eStream application server can handle 1340.

Assumptions Underlying Server Scaling Estimate

User's application usage pattern [what is run, for how long, what features] does not change materially depending on whether s/he is using eStream 1.0 or Personable/Citrix. It would be interesting to collect data on typical usage patterns; Sridhar formulated estimates for his network bandwidth evaluation, but it might be interesting to see if our in-house use of common applications matches those estimates.

Overall application server capacity is adequate for both products. Personable/Citrix may deny application server access to clients when insufficient overall application server capacity is available, whereas eStream 1.0 may continue to allow clients to access the

servers without some explicit client capacity cutoff point, given the usual small impact of each additional eStream client. However, it is possible that an extreme performance collapse could occur on eStream, if client load were to grow so large compared with the capacity of eStream's application servers that the lack of server response caused an escalating number of redundant requests from eStream clients retrials. This situation needs to be avoided, in the eStream 1.0 design and/or in its deployment.

The increased client servicing capacity afforded an eStream 1.0 application server will not uncover some insurmountable system bottleneck never encountered with the limited client servicing capacity possible on the Personable/Citrix server, including areas such as maximum number of threads, processes, sockets, buffer size for socket send or receive, socket listen queue length, network buffer cache, maximum number of file handles, etc.

eStream 1.0 clients are fat enough to allow adequate client caching to hit or better the client cache miss goal of 2%. Thin clients (which are not the intended design target for eStream 1.0) or fat clients with inadequately sized client caches would increase server overhead for paging requests & network traffic beyond the levels estimated in this document.

Design Goals Supporting Server Scaling Estimate

Overall client cache miss rate is less than 2%. Reaching the eStream 1.0 client performance goals also reinforces the need for a low client cache miss rate. We have not collected data indicating how large a client cache would be needed to hold application pages and file metadata associated with typical application usage as represented by the Ziff-Davis benchmark runs. I think it would be useful to have such data, since it may influence client cache design & effective cache management policies.

AppInstallBlock server overhead is no more than the equivalent of an extra 0.5% cache miss rate. Installation is expected to be relatively rare, & downloaded material is expected to be kept at the minimum size necessary to reach our functionality & client performance goals.

Client wasted prefetch overhead is no more than the equivalent of an extra 0.5% cache miss rate. With fat clients & large warm caches, prefetching is expected to be kept at a minimum for eStream 1.0; again, data gathering related to this area may be useful.

eStream File System Straw Man Proposal

Version 0.5

[redacted]

Purpose

The purpose of this document is to present a concrete proposal for the functioning of the eStream file system. In many places, I make some sweeping generalizations about how things should work without describing the data structures and interfaces involved in implementing them. This document should eventually involve into a design specification.

Issues Not Covered

This document does not attempt to cover all issues present in designing the eStream 1.0 product. In particular, the overall authentication/licensing/security architecture is not covered in detail here. It is expected that the security functionality will be mostly orthogonal to the design of the basic file system functionality.

Background

There are a number of different networked file systems out there. Many of them share some requirements with eStream. For example, AFS performs client-side on-disk caching, while Coda handles serious server redundancy and disconnected operation. Personally, I believe that AFS and Coda are the file systems whose designs are most relevant to us. For those interested in further background reading, you might also want to look at papers covering NFS, CIFS, xFS, DFS, and Zebra.

Single File System Name Space

Many modern distributed file systems present the network file system as a single tree mounted at some location on the client system, regardless of which server hosts the data. (In fact, with AFS, every file on every server in the world can be accessed through a path starting with /afs on the client, assuming the client can reach that server and has sufficient privileges to do so.) Compared with systems like NFS and Windows sharing, where each share is mounted in a different location on the client, the single name space provides greater ease of use.

The eStream file system would present one universal logical file system. Regardless of which ASP provider supplies a particular volume, that volume will always be referenced via the same path on the eStream file system. That this is desirable or even feasible is predicated on the assumption that OTI is the only entity providing all eStream sets. Each volume must get a unique identifier and a unique location to be mounted in the file system hierarchy. If two different ASPs privide the same volume ID, then the contents of those volumes must be identical. This way, we don't have to tag things in the cache based on what ASP they came from, and the cache manager doesn't need to know anything about ASPs. If done correctly, only the client networking component and the LSM need to know about ASPs.

Volumes

A volume is a complete subtree of a file system. Volumes may contain files and directories. Volumes may not be mounted in other volumes. A volume is a logical grouping of files within the file system and is the unit of replication across servers. An application will reside in a single volume. Two applications will never share a volume.

Volumes are uniquely identified by a 32-bit volume identifier. Each volume additionally has an 8-bit version number. This version number is incremented each time any file within the volume changes. (See supporting upgrades, below). Note that the volume id is globally unique. If two ASPs provide volumes with the same volume number (and version), they have identical contents.

A volume may be replicated on any number of servers. Each SLM server contains a map describing the application servers that currently provide each volume. This global replication of this table is acceptable because volumes are added or moved infrequently.

Identifying Files

Files and directories are uniquely identified by the pair (volume id, file number). This tuple is called a file id. Volume id and file number are each 32-bit signed integers. Negative values for both volume id and file number are reserved for special purposes, leaving us with 2^31 possible volume IDs and 2^31 possible files per volume.

Finding an Application Server for a Volume

The SLM will tell the client which application servers currently provide each volume. It may be necessary for the client to periodically poll the SLM to get up-to-date information about the state of the application servers. The License and Subscription Manager on the client will keep track of the currently subscribed applications and the application servers for each of these applications.

Directories

Directories are specially formatted files that are used in a special way by the file system. They are identified by file ids, just like other files. From a client-server point of view, they are read by the client in the same way as other files. Directories contain arrays of entries with the following format:
(volume number, file number, flags, length, filename)

The volume number and file number are 32-bit signed integers. The flags are 32-bits of flags. The length is 16 bits and is the length of the filename in bytes. The filename is a non-NUL terminated Unicode string. The structure is padded with enough Unicode NUL characters to make the structure a multiple of 32 bits long. The next directory entry begins on the next 32-bit boundary.

The access token is not part of the directory, as a single access token is required to access all files in a particular volume.

The volume number is required so that the the client can construct a local directory for the root of the directory structure in the same format as other directories (see filename parsing below). It also helps to provide a sanity check.

Accessing Files

Assuming that a client has a file-id for a file that it wishes to access, the following client-server actions must be supported:

For stat-like information on the file, we need a GetFileMetadata() interface. The client would provide the file id it is interested in and the proper access token for this file. The server will either return the metadata for the file or an error condition (like access token expired or incorrect access token.) The metadata contains the standard Windows metadata information, including file length and file access times.

On a file open (CreateFile in Windows terminology), we need to verify that we have access to the requested file. This is probably best accomplished by calling GetFileMetadata and verifying that we can get the metadata. This way, we can fail file opens gracefully if we don't have an access token.

On reads (and writes, when we support them), the client will send the file id and the access token to the server along with an offset and a length for the read and write. The server will respond with the data. Note that the same mechanism will be used for reading both files and directories.

Pseudodirectories

For those parts of the eStream file system name space that do not belong to any volume (such as the root of the file system), the client must construct appropriate directories based on the currently installed applications. This is to support filename parsing starting at the root of the directory. For example, if the client has word installed with a root of /Worddir and it is volume number 3 and Photoshop installed with a root of /Photodir and is volume number 4, the client would construct a directory for the root of the entire file system containing
File name, Volume number, file number
"Worddir", 3, 0
"Photodir", 4, 0

(The file numbers are both zero here because 0 is the index of the root directory of each volume, and these are the mount points for each volume.)

When new applications are installed, the root of the file system would have to be updated to reflect the newly installed apps.

Filename Parsing

Filename parsing is handled one element at a time, starting at the root of the file system. Parsing one path name element involves reading the parent directory's contents (from the

cache or the app server), searching it for the file matching the next path element's name, and getting the appropriate file id so it can do further lookup.

Volume Versioning... Without File Versioning

We can provide volume versioning and incremental volume updates without versioning each file in the file system. When a new volume is to be provided, we can append any new or changed files as new files in the volume, with new volume IDs that weren't already present. If a directory's contents have changed, then a new version of this directory will be built, with a new file number. This process will proceed from the leaves all the way to the root of the file system, eventually resulting in a new root. The old versions of things would still be available for old clients to access, but clients wishing to access the new version will simply start at the new root, and would thereby get to a consistent picture of the volume. Any file or directory that has not changed from the old version to the new one need not be replicated, and will be referenced by its old file number. (I.e. newly reconstructed directories will contain the old file number for any files that haven't changed.)

If we reserve the first 256 file ids for the root directory, then the version number can be the same as the file number for the root directory.

Note that if we decide that the complexity of this approach is too high, this does not preclude always creating a new volume from scratch for each update.

Constructing File IDs

It is the job of the builder to produce the volume file to file id mapping and to construct all of the directories. Because directories are files identified by file id, this process must begin at the leaves of the volume and proceed to the root.

Note that constructing a new changed volume will consist of finding the diffs between the two volumes and producing some new directories. Changed or newly added files will get new file numbers, leaving the old ones around. Note that any directory that has had any descendents changed must be reconstructed with the new file numbers, and the new directory will get a new file number. This process will proceed to the root of the volume, which will receive a new file number.

Server Failover

All app servers for a particular volume must share the same mapping of file ids to file, so server failover is trivial. There might be a performance impact if the new app server doesn't have the requrested file in memory.

Writing Files into the Application Install Directories

Two approaches have been discussed for the problem of applications that want to write files to their install directories. First, this can be handled wholly inside of the eStream file system. The cache manager could allow writes to files handled by the eFS, but these writes would not be written back to the server. Instead, they would simply be written to

the eFS cache and marked non-purgeable. This approach's primary advantage is that it does not rely on a file spoofer.

The other approach is to use the file spoofer to spoof some accesses to the z: drive. Any open for read/write access would cause the existing file (if any) to be copied to a location on the c: drive, and the file spoofer would then redirect the open to the newly created file. The file spoofer would have to keep track of any file created via this copy-on-write mechanism and redirect all future accesses to the copy. There are some issues to this approach. For example, it is extremely wasteful when files on the z: drive are opened for read/write access but are never actually written. However, it does help reduce the complexity of the eFS cache, and is trivial to implement if we have to do c: to z: file spoofing anyway.

In either case, to support the creation of new files in an application's install directory, it must be possible to modify the contents of directories in the cache.

If we don't use the file spoofing approach, there is the issue of how we support written files when we move to a newer version of a volume. It would probably be necessary to walk the cache and make sure that each written file gets placed in the appropriate place in the new volume version. This is likely to be non-trivial, because we need to have full information about the location of each modified file in the file system tree, and would need to download enough of the new volume directory structure to place these modified files there.

64-Bit File Access?
One question we should answer is whether we will support file sizes greater than 2 GB on the eStream file system. I'm inclined to say that such support isn't a requirement for the 1.0 product, but I also think that the implementation and verification complexity of 64-bit file access on the file system is low enough that we might want to consider building it in anyway.

Simplifications
We could preclude the possibility of an application consisting of more than one volume.

Future Possibilities
Epicon seems to make a big selling point of their technology involving "self-healing" of damaged application files. Such support could be provided by computing checksums on files in the cache. Whether or not we want to support this is an open question. My feeling is that it's something we should leave out of 1.0.

Outstanding Issues
Cache organization has not been addressed.

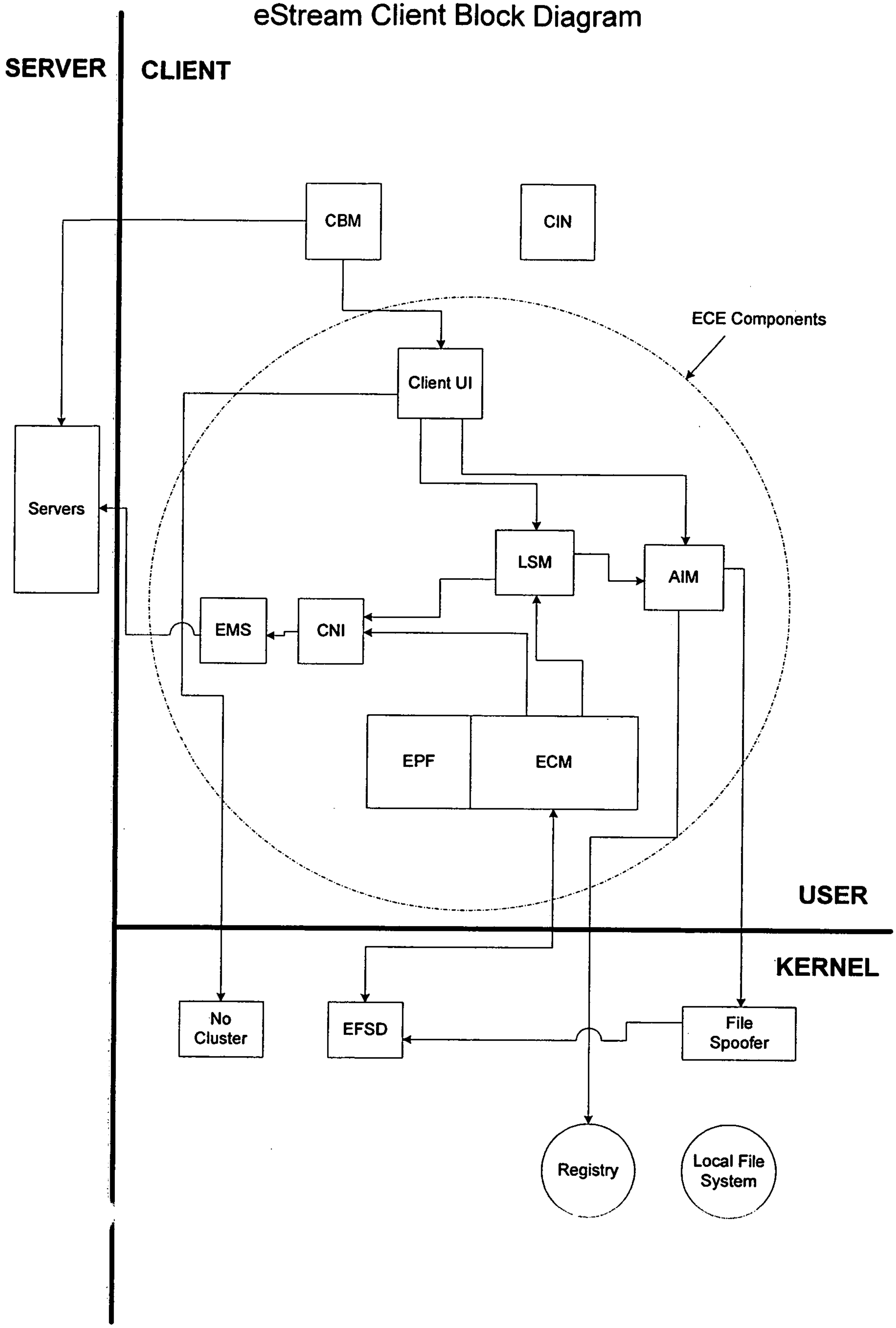
Finding and downloading the app install block has not been addressed.

Security in a multiuser system has not been addressed.

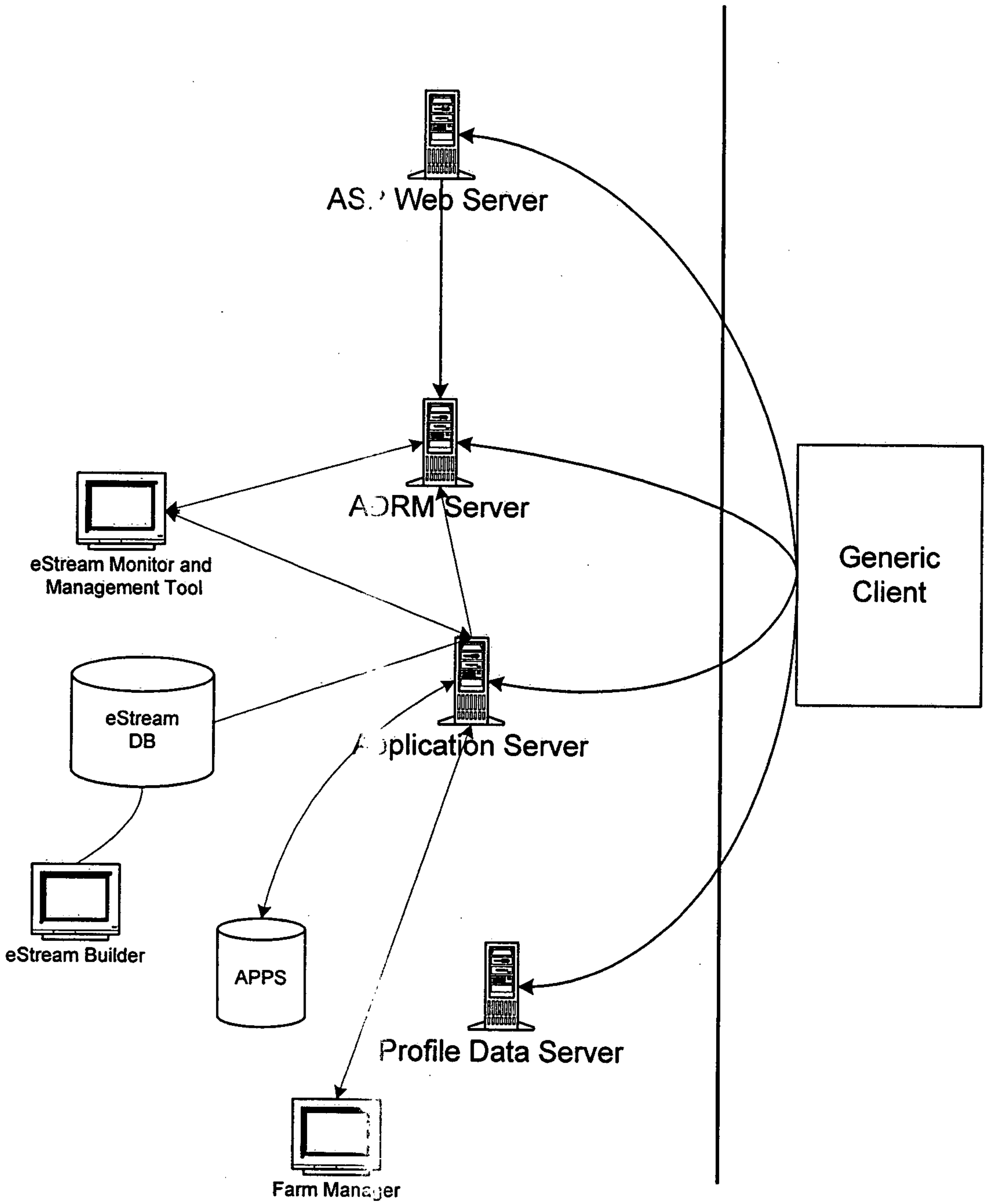
eStre:am Builder Block Diagram

???

eStream Client Block Diagram



eStream Server Block Diagram



eStream Requirements from Multiple Perspectives

Ricky Benitez
[redacted]

1.0 OTI Requirements

The following requirements are primarily driven by OTI and its business and operational needs.

1.1 Functionality

- The system can deliver applications to x86-based Windows NT 4, Windows 2000, Windows 95, Windows 98 and Windows Me systems clients in order to capture a substantial size of the existing client market
- The client and server software, software upgrades and application sets can be delivered to customers electronically via the internet (ftp or web) to reduce the costs associated with the delivery of the product
- The client and server software, software upgrades and application sets can be delivered to customers on traditional media (CDs, DVDs) to increase the speed of service at remote sites and to provide the flexibility of servicing isolated networks
- The system allows Omnishift to perform audits, with the permission of the ASP, to ensure that we are being appropriately paid per our license agreement with the ASP

1.2 Localization

- The language used by the server components should be specific to an administrator to facilitate the installation and remote operation of a foreign-run server by OTI personnel

1.3 Usability

- The system must incorporate troubleshooting facilities that allow the support organization to identify and fix issues without engaging the engineering team
- The process of converting applications to eStream sets and testing those eStream sets (certification) must be fairly expeditious (target an average no more than 1 person's effort per application per day)

1.4 Reliability

- The internet-based software, software upgrade and application delivery system must be highly available to support ongoing remote installation and service work

1.5 Performance

- The system must be no less than 50X more scalable than Windows Terminal Server/Citrix Metaframe when running client applications to make it a cost-effective high-volume client application delivery mechanism

1.6 Scalability

- Given sufficient hardware and network resources, the internet-based software, software upgrade and application delivery system must scale to handle any potential number of customers
- The system can be easily expanded to support 64-bit file sizes
- The system provides 128-bit application identifiers
- The system supports applications that are composed of up to 2 billion files

1.7 Security

- It must be difficult enough to steal an application's code and data from the client that software providers are not concerned about using the system as a delivery mechanism
- It must be very difficult to gain unauthorized entry to the internet-based software, software upgrade and application delivery system
- It must be computationally infeasible to steal data or code while it is being distributed across the internet-based software, software upgrade and application delivery system
- It must be computationally infeasible for the internet-based software, software upgrade and application delivery to be coerced by a third-party into delivering a Trojan
- The internet-based software, software upgrade and application delivery must log all accesses so that an appropriate security audit can be periodically performed

1.8 Portability

- The server components must be portable across a wide variety of platforms and operating systems, including but not limited to: Windows NT 4, Windows 2000, Solaris UltraSPARC, HP-UX, and Linux

1.9 Maintainability

- The entire system must be component-wise upgradeable without requiring an interruption of service by any of our customers
- The server components provide a web-based monitor and administrative console that can be used to diagnose and maintain an ASP site
- The client components provide an interface that can be used to diagnose client problems and provide customer support
- All application upgrades are backwards compatible

2.0 ASP Requirements

The following requirements are primarily driven by an application service provider's business and operational needs.

2.1 Functionality

- The system can deliver client applications (e.g., productivity applications such as Microsoft Office, collaborative applications such as email and groupware, content creation applications such as Adobe Acrobat and Photoshop and client front end applications for client/server and 3-tier enterprise applications such as the People Soft client). Certification of additional client applications will require an additional certification fee. Omnishift does not guarantee that any given application will successfully certify even if a previous version of the application was certified
- The following application eStream sets will be certified for the initial release of the product (the latest available versions should be assumed unless otherwise indicated):
    - Lotus 1-2-3
    - Lotus Word Pro
    - Microsoft Access
    - Microsoft Powerpoint
    - Microsoft Word
    - Microsoft Excel
    - Microsoft Outlook
    - Microsoft FrontPage
    - Adobe Photoshop
    - Adobe Premiere
    - Adobe Acrobat
    - Sonic Foundry Sound Forge
    - Macromedia Director
    - Macromedia Dreamweaver
    - Intuit Quicken
    - Intuit Turbotax
    - Qualcomm Eudora
- The system provides facilities for reporting application usage and usage patterns
- The system supports and reasonably enforces the following license payment models:
    - One-time charge for unlimited use
    - Pay per month
- The system supports and reasonably enforces the following license models:
    - Unlimited simultaneous use
    - No simultaneous use on multiple clients

- The system supports the ability to have the user view and agree to license terms upon subscription of a license
- The system can deliver applications to any client capable of connecting to the internet over a standard HTTP connection
- The system can service customers behind corporate and personal firewalls
- The system can be integrated to work with any existing billing system and database, although a consulting and customization fee will be charged for integration with any database other than SQL server and any billing system other than ?
- The system provides an external API that makes it easy to interface with standard billing solutions
- The system provides capabilities for reviewing billing information at the global, account and user levels
- The system provides user management (create, edit, remove, enable, disable, remove, add to group, remove from group) capabilities including user groups
- The system provides account management (create, edit, remove, enable, disable) capabilities that are integrated with the accounts in the billing system (a 1-to-1 relationship between billing accounts or sub-accounts and system accounts)
- The system provides application management (create, edit, enable, disable, remove, add to group, remove from group) capabilities including application groups
- The system provides subscription and un-subscription capabilities that allow a user group access to an application group under a supported license model
- The system provides capabilities for automatically upgrading an application upon next use, for allowing a client to optionally upgrade on each subsequent use or for informing a client that an upgrade is available for subscription on each subsequent use

2.2 Localization

- The server components and web interface components must be able to operate across the different languages supported by Windows, although only American English menus, panels, messages, help and documentation will be initially made available

2.3 Usability

- The system must not require any scheduled or unscheduled down time
- The system provides appropriate help panels to the system administrator

2.4 Reliability

- Given sufficient hardware resources and an appropriate configuration, the system must be able to tolerate the failure of any one server either hosting the application or the software license management service without interrupting the service of any current or future clients

- The system should tolerate single failures in which a server or client transmits bogus messages or fails to transmit expected messages

2.5 Performance

- The system's overall price/performance makes it profitable to deliver low-cost client applications at volume levels
- The system must be able to operate satisfactorily with an average bandwidth of 256Kb/s or more
- The system must be able to operate satisfactorily with an average latency of 0.25 seconds or less
- The system allows encryption to be enabled or disabled either globally or on a domain basis to improve system performance inside protected LAN environments
- The system uses bandwidth discovery to determine whether compression is beneficial when sending massages between the client and the server

2.6 Scalability

- A fully-functional, non-redundant server system can be set up and configured to run on a single server machine
- The system must be scalable, given sufficient machine resources, to support any number of clients and certified applications
- The system can make effective use of the servers available to serve applications and license tokens without the need for constant administrative supervision and management

2.7 Security

- The server must be able to operate behind an industry-standard firewall solution
- The system must withstand DOS (Denial Of Service) attacks with only localized degradations in service
- The system must allow diagnostic modes to be locked out or disabled to ensure that no outside party, including OTI, can log on and administer the server components
- The system allows for different levels of account, billing, user and subscription access so that the various capabilities can be allowed or denied on a user or user group basis
- The system logs all application usage activity to a specific user and client so that appropriate security and usage audits can be performed

2.8 Portability

- The server hardware platform does not limit the platforms that applications can be delivered to from that server

2.9 Maintainability

- The system must incorporate troubleshooting facilities to ease the burden of identifying, isolating and resolving operational issues
- The server components must be able to log all abnormal activity and the administrator should be able to select which activity should be logged and which ignored
- The server supports SNMP queries so that it can be integrated with a system management solution like Openview
- The server must be easily configured so that alarms and alerts can be defined when important events occur
- New applications and application upgrades can be installed without interrupting the current use of that application or any other applications by any client
- It must be easy to add, remove, enable and disable application and license servers without affecting the rest of the system, requiring the system to be shut down or affecting application delivery to any current or future client (provided that sufficient alternative servers are available to maintain operations)

3.0 Client User Requirements

The following requirements are primarily driven by a client user needs and expectations on the system.

3.1 Functionality

- The system supports multiple applications running on the same client, including multiple instances of the same application (within reasonable limitations of the operating system, the client hardware and the applications)
- The system allows a single client to be simultaneously connected to multiple application service providers
- The system provides the capability to view and edit billing, account, subscription and user information from a connected client
- The system supports user roaming so that a user can access their applications from any machine that has installed the client and can access an application service provider over the network
- The system allows user data to be saved on the local machine
- The system allows applications to print to any local and network printers accessible to the client machine
- The system allows any set of applications to interact with each other via OLE automation, COM and signals regardless of whether the applications are installed locally or delivered by the system
- The system can be deactivated and reactivated by the user on demand
- The system can be set up to work through a proxy server

3.2 Localization

- The client components must be able to operate across the different languages supported by Windows, although only American English menus, panels, messages, help and documentation will be initially made available

3.3 Usability

- The average interactive response time of an application delivered by the system will not exceed 110% that which would be experienced by the user were running a locally-installed version of the application
- The system client components are easily downloaded and installed
- The system client components are easily uninstalled
- The system detects the presence of a local version of the application being installed and warns the user that their local version will become unavailable while the system is operational
- The system allows a pre-existing local version of the application to become available when the system's version of the application is uninstalled
- The system does not require a reboot of the client whenever a new application is installed or uninstalled
- The system removes all application-related traces from the client system when an application is uninstalled
- The system must be able install itself and operate on a client with only 16MB of disk space available, although the system may request that the user free up and reserve more disk space to improve the performance of the system
- The user can, through an advanced options menu, manage the size of the local system cache if they choose to override the system default behavior
- The system provides the user access to information regarding cache usage, bandwidth usage and connectivity status
- The user must be able to control which license to use when it can obtain a license from more than one connected application service provider
- The system provides appropriate help panels to the user
- The system allows a user to indicate that they do not wish to be informed of an optional upgrade again for a particular application and respects their choice on that matter
- The system quickly releases non-simultaneous use licenses after their use so that the same user can use the license from another client
- The system release non-simultaneous use licenses within a short span of time after a client that was using such a license crashes or is unexpectedly shut down
- A client is given sufficient notification of an impending license expiration to save their work and take appropriate action to renew or extend the license
- Upon an expiration of a license, the application is halted and the user given the ability to renew the license. The application is never terminated by the client components unless the user gives their consent to an actual termination

3.4 Reliability

- The client components can run indefinitely as they consume only a managed amount of system resources
- The system must deliver code and data to the client identical to that it would have were the application residing locally
- The client component can "heal" itself from deleted client dlls

- The system detects and recovers from garbled network messages although at potentially reduced performance

3.5 Performance

- The client must make a run/no run license decision quickly for any application execution request
- The system collects profile information locally so that future accesses are tailored to local usage patterns

3.6 Scalability

- There must be no system-imposed limits on how many applications can be subscribed, installed or executed simultaneously

3.7 Security

- All subscription, financial and credit card transactions are performed securely making it computationally infeasible for a third party to obtain any such information
- It must be computationally infeasible for the system to be coerced by a third-party into delivering a Trojan
- It must be computationally infeasible for a third party to determine which application a client is executing
- You must be able to specify that you do not wish your application usage information to be made available to any third party

3.8 Portability

- The system automatically delivers the appropriate version of a subscribed and installed application to the client provided that an appropriate certified version of the application is available at the application service provider's site for the client hardware and operating system configuration

3.9 Maintainability

- The client components should not require any user administration
- The system client components should be upgradeable without requiring a re-installation or re-subscription of existing applications

readme.txt arai [redacted]

This directory /docs is for eStream design documentation only.

This directory is reserved for high level documents such as the eStream high level requirements and design documents.

The subdirectories "client," "server," and "builder" are for documents specific to those pieces of eStream. These directories will be used to contain component-wide documents, such as the server component framework and the eStream set format document. Each directory will contain subdirectories for each sub-component. The names of the subcomponent directories will be the abbreviation or acronym for the component, in all lower case. Some client components will be ecm (cache manager), efsd (file system driver), epf (prefetching and fetching), and cni (client network interface).

Another subdirectory of "documents" will be "eng". This will include all company-wide engineering documents, such as coding guidelines and design document templates.

Document and directory names will contain no spaces. (These interoperate poorly with non-Microsoft platforms.) Files will contain the the name of the component followed by the acronym for the document type, separated by a hyphen '-'. Recognized document types are currently:
LLD - Low level design
REQ - requirements
HLD - High level design
SM - strawman

Thus some documents would be
CacheManger-LLD.doc (low level design for the cache manager)
PrefetcherFetcher-SM.doc (profiler/prefetcher component strawman)

Estream 1.0 Planning Document

[redacted]

Low-Level Design Status/Plan

Sub Components	Owner	LLD Design Doc completed	LLD review Complted	Estimates for Impl	Impl and Unit Test Completed
Content					
Install Monitor	Sanjay	Done	Done	3 wk	
Builder GUI	Sanjay	Done	Done	1 wk	
FSRFD (Drivers)	Sanjay	Done	Done	2 wk	
AppInstallBlk structure	David	Done	Not needed		
Profiler	David	Done	Done	2 wk	
File Access Monitor	David	Done	Done	1 wk	
Packager	David	Done	Done	1 wk	
eStream distribution	**Bob**	**8/31/2000**	**Status TBD**	**TBD**	
Server Group					
Web Server	Bhaven	Done	Done	8 wk	
Monitor	Mike	Done	Done	4 wk	
SLiM Server	Amit	Done	Done	2 wk	
App Server	Sameer	Done	Done	4 wk	
Admin UI	Bhaven	TBD	TBD	TBD	
End User UI	Bhaven	TBD	TBD	TBD	
Common Server Components	Mike	Done	Done	3 wk	
Messaging	Sameer	Done	Done	3 wk	
Threads Package	Sameer	No Document		1 wk	
Security Design	**Igor/Amit**	**Not Done**	**Not Done**	**TBD**	
Client Group					
Cache Prefetching	Anne	Done	Done	1 wk	[redacted]
LSM + Plug in	Anne	Done	Done	1 wk	
Client UI	Anne	Done	Done	1 wk	
Client Installer	Anne	Done	Done	1 wk	
Start Client	Anne	Done	Done	1 wk	
Application Install Mgr	Nick	Done	Done	TBD	
Piracy	Nick	Done	Done	TBD	
File Spoofer	Curt	Done	Done	1 wk	
eStream File System	Curt	Done	Done	8 wk	
NoCluster Driver	**Curt**	**Status TBD**	**Starus TBD**	**2 days**	
eStream Cache Manager	Dan	Done	Done	8 wk	
Client Network Interface	Dan	Done	Done	2 wk	

Implementation Plan

Milestones

[redacted]

ECM (RAM disk cache) and EFSD executes a local "himom" executable
Photoshop is installed locally and successfully executed from estream sets and appInstallblk produced by builder
App Server and EMS integrated to copy "himom" executable using a dummy client
App Server, EMS and CNI integrated to copy "himom" executable from "himom" estream sets
office is installed locally and successfully executed from estream sets and appInstallblk produced by builder
App Server, EMS, ENI, ECM and EFSD integrated to run "himom" from estream sets on server
Following applications built and tested with local installation

Adobe Premier
Macromedia Director and Shockwave
Corel Suite
Lotus Suite

[redacted] **Photoshop is installed by AIM and executed from estream sets on App server**

No Subscription
No License Management
RAM cache for ECM
Installation of Photoshop using AIM

[redacted] **Photoshop is installed by AIM and executed from estream sets on App server**

No Slim Server
Disk based cache for ECM

Estream includes initial prefetched pages and these pages are prefetched during installation
Fully functional estream bits (includes initial prefetched pages)
Client software is run as a service
App Server is started by Monitor
Admin UI to stop and start app Server
Application subscription from web server
installation on client after subscription

[redacted] Testing environment is setup (configuration of 3 servers and one client)

[redacted] Photoshop runs with the following additional functionality

Leads for milestone: Amit and Nick
Slim Server
http protocol
CNI supports uique message ids for NAD
Fully functional LSM
Real Accesstokens
Uninstall applications
Anti-Piracy support
AppServer and SlimServer fail-over
File spoofing

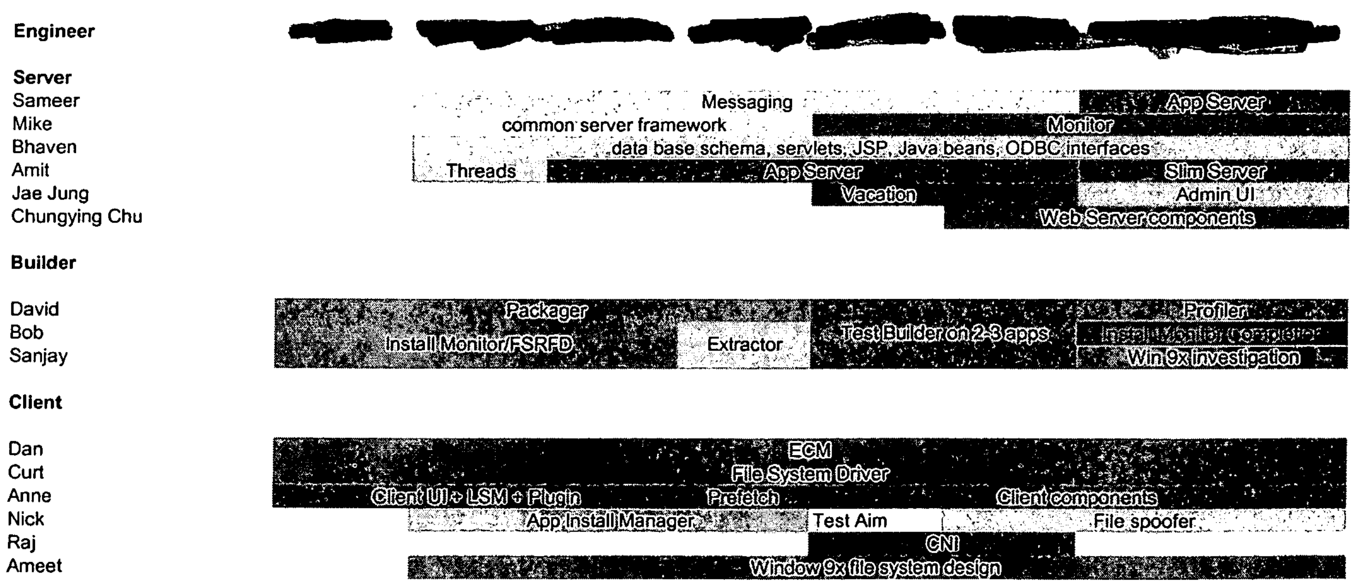
[redacted] Clean builds by integration (George) (Raj will drive this)

[redacted] Office is running with full functionality

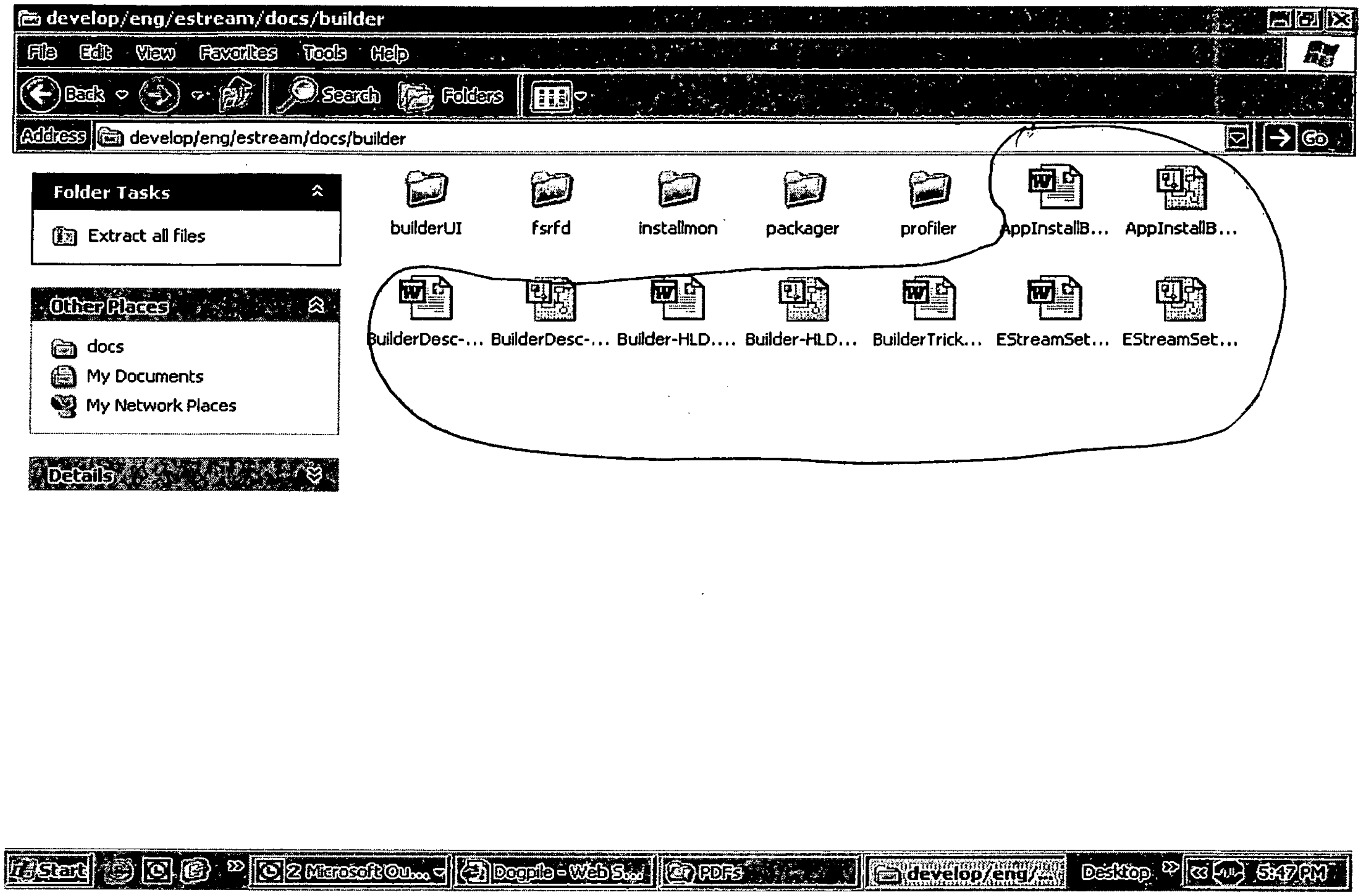
Restruturing of client so it can be started at boot time
Performance tuning
Improve robustness
application upgrade
Crash resiliency
All software purified and memory leaks eliminated
(May be) Applets for monitoring server components

[redacted] **Office is removed from desktop of at least one person and reinstalled using estream**

[redacted] Code Freeze



Builder


develop/eng/estream/docs/builder
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/builder
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD....
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...
Start
2 Microsoft Ou...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
5:47 PM

eStream AppInstallBlock Low Level Design

Sanjay Pujare and David Lin
Version 0.5
[redacted]

Functionality

The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the eStream client to 'initialize' the client machine before the eStream application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that eStream application.

The AppInstallBlock is created offline by the eStream Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system, and any files added or modified in the system directories. Files added to the application specific directory is not recorded in the AppInstallBlock to reduce the amount of time needed to send the AppInstallBlock to the eStream client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

The AppInstallBlock and the runtime data are packaged into the eStream Set by the Builder and then uploaded to the application server. After the eStream client subscribed to an application and before the application is run for the first time, the AppInstallBlock is send by the server to the client. The eStream client invokes the default initialization procedure and the optional application-specific initialization code. Together, the default and the application-specific initialization procedure process the data in the AppInstallBlock to make the machine ready for eStreaming that particular application.

Data type definitions

The AppInstallBlock is divided into the following sections: header section, variable section, file section, profile section, prefetch section, comment section, and code section. The header section contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section is a tree structure consisting of the files copied to C drive during the application installation. The profile section contains the initial set of block reference sequences during

Builder profiling of the application. The prefetch section consists of a subset of profiled blocks used by the Builder as a hint to the eStream client to prefetch initially. The comment section is used to inform the eStream client user of any relevant information about the application installation. Finally, the code section contains an optional program tailored for any application-specific installation not covered by the default eStream application installation procedure. In Windows version, the code section contains a Windows DLL.

Here is a detailed description of each fields of the AppInstallBlock.

Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4K byte size.

1. Header Section:

The header section contains the basic information about that AppInstallBlock. This includes the versioning information, application identification,

Core Header Structure:

- **AibVersion [4 bytes]**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).
- **AppId [16 bytes]**: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.
- **VersionNo [4 bytes]**: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.
- **ClientOSBitMap [4 bytes]**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future OSs we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- **ClientOSServicePack [4 bytes]**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap.
- **Flags [4 bytes]**: Flags pertaining to AppInstallBlock
    - **Bit 0:** Reboot – If set, the eStream client needs to reboot the machine after installing the AppInstallBlock on the client machine.
    - **Bit 1:** Unicode – If set, the string characters are 2 bytes wide instead of 1 byte.
- **HeaderSize [2 bytes]**: Total size in bytes of the header section.
- **Reserved [32 bytes]**: Reserved spaces for future.

- **NumberOfSections [1 byte]**: Number of sections in the index table. This determines the number of entries in the index table structure described below:

Index Table Structure: (variable number of entries)

- **SectionType [1 bytes]**: The type of data describe in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.
- **SectionOffset [4 bytes]**: The offset from the beginning of the file indicates the beginning of section.
- **SectionSize [4 bytes]**: The size in bytes of section.

Variable Structure:

- **ApplicationNameLength [4 bytes]**: Byte size of the application name
- **ApplicationName [X bytes]**: Null terminating name of the application

2. File Section:

The file section contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into 'unusual' directory during the installation of an application. If the file content is small, the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is list of trees stored in a contiguous sequence of address space according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directory. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

Directory Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 0 is set if this entry is a directory
- **NumberOfChildren [2 bytes]**: Number of nodes in this directory
- **DirectoryNameLength [4 bytes]**: Length of the directory name
- **DirectoryName [X bytes]**: Null terminating directory name

Leaf Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 1 is set to 1 if this entry is a spoof or copied file name
- **FileVersion [4? bytes]**: Version of the file GetFileVersionInfo() if the file is win32 file image. Need variable file version size returned by GetFileVersionInfoSize(). Otherwise use GetFileTime() to retrieve the file creation time.
- **FileNameLength [4 bytes]**: Byte size of the file name

- **FileName [X bytes]**: Null terminating file name
- **DataLength [4 bytes]**: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file
- **Data [X bytes]**: Either the spoof file name or the content of the copied file

3. Add Variable and Remove Variable Sections:

The add and remove variable sections contain the system variable changes needed to run the application. In Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address space according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

a. Registry Subsection:

1. "HKCR": HKEY_CLASSES_ROOT
2. "HKCU": HKEY_CURRENT_USER
3. "HKLM": HKEY_LOCAL_MACHINE
4. "HKU": HKEY_USERS
5. "HKCC": HKEY_CURRENT_CONFIG

Tree Structure: (5 entries)

- **ExistFlag [1 byte]**: Set to 1 if this tree exist, 0 otherwise.
- **Key or Value Structure entries [X bytes]**: Serialization of the tree into variable number key or value structures described below.

Key Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **NumberOfSubchild [4 bytes]**: Number of subkeys and values in this key directory
- **KeyNameLength [4 bytes]**: Byte size of the key name
- **KeyName [X bytes]**: Null terminating key name

Value Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **ValueType [4 byte]**: Type of values from the Win32 API function RegQueryValueEx(): REG_SZ, REG_BINARY, REG_DWORD, REG_LINK, REG_NONE, etc...
- **ValueNameLength [4 bytes]**: Byte size of the value name
- **ValueName [X bytes]**: Null terminating value name

- **ValueDataLength [4 bytes]**: Byte size of the value data
- **ValueData [X bytes]**: Value of the Data

In addition to registry changes, an installation in Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the eStream client machine. The ini entries are appended to the end of the variable section after the 5 registry trees are enumerated.

b. INI Subsection: (not supported in eStream 1.0)

- **NumFiles [4 bytes]**: Number of INI files modified.

File Structure: (variable number of entries)

- **FileNameLength [4 bytes]**: Byte length of the file name
- **FileName [X bytes]**: Name of the INI file
- **NumSection [4 bytes]**: Number of sections with the changes

Section Structure: (variable number of entries)

- **SectionNameLength [4 bytes]**: Byte length of the section name
- **SectionName [X bytes]**: Section name of an INI file
- **NumValues [4 bytes]**: Number of values in this section

Value Structure: (variable number of entries)

- **ValueLength [4 bytes]**: Byte length of the value data
- **ValueData [X bytes]**: Content of the value data

4. Prefetch Section:

The prefetch section contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing for frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of blocks to include in the prefetch section is the blocks associated with frequently accessed directories and file metadata in this directory. The prefetch section is divided into two subsections. One part contains the critical blocks that are used during startup of the streamed application. The second part consists of the blocks accessed for common user operations like opening and saving of document. The format of the data is described below:

a. Critical Block Subsection:

- **NumCriticalBlocks [4 bytes]:** Number of critical blocks.

Block Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

b. **Common Block Subsection:**

- **NumCommonBlocks [4 bytes]:** Number of critical blocks.

Block Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

5. Profile Section: (not used in eStream 1.0)

The profile section consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry *[row, column]* of the matrix is the frequency a block *row* is followed by a block *column*. In any realistic applications of fair size, this matrix is very large and sparse. Proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the eStream client prefetcher performance can be increased.

Row Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the row block
- **BlockNumber [4 bytes]**: Block Number of the row block
- **NumberColumns [4 bytes]**: number of blocks that follows this block. This field determines the number of column structures following this field.

Column Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the column block
- **BlockNumber [4 bytes]**: Block Number of the column block
- **Frequency [4 bytes]**: frequency the row block is followed by column block

6. Comment Section:

The comment section is used by the Builder to describe this AppInstallBlock in more detail.

- **CommentLength [4 bytes]**: Byte size of the comment string
- **Comment [X bytes]**: Null terminating comment string

7. Code Section:

The code section consists of the application-specific initialization code needed to run on the eStream client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the eStream client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: *Install*(), *Uninstall*(). The eStream client loads the DLL and invokes the appropriate function calls.

- **CodeLength [4 bytes]**: Byte size of the code
- **Code [X bytes]**: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

8. LicenseAgreement Section:

The Builder creates the license agreement section. The eStream client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

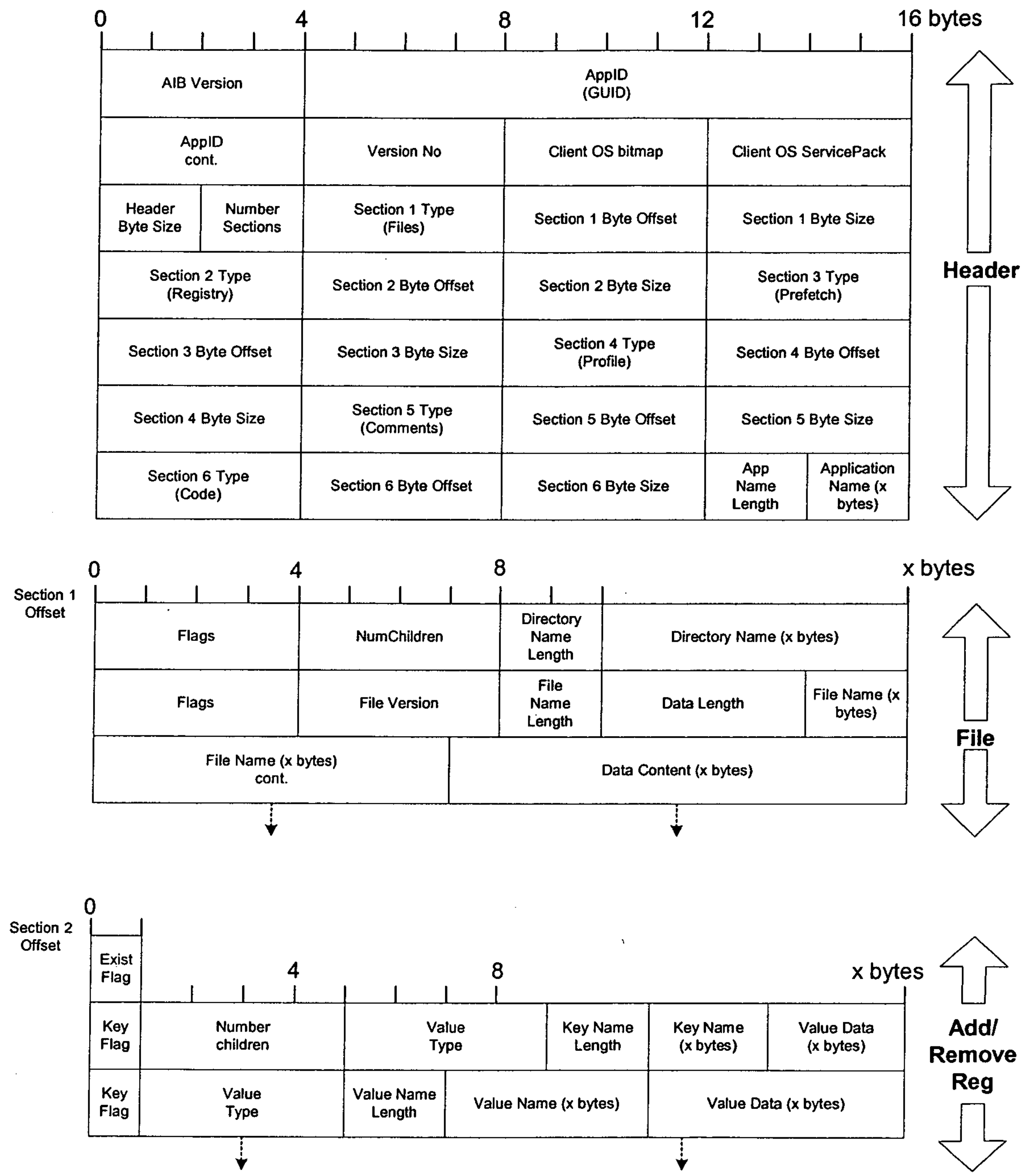
- **LicenseTextLength [4 bytes]**: Byte size of the license text
- **LicenseAgreement [X bytes]**: Null terminating license agreement string

Open Issues

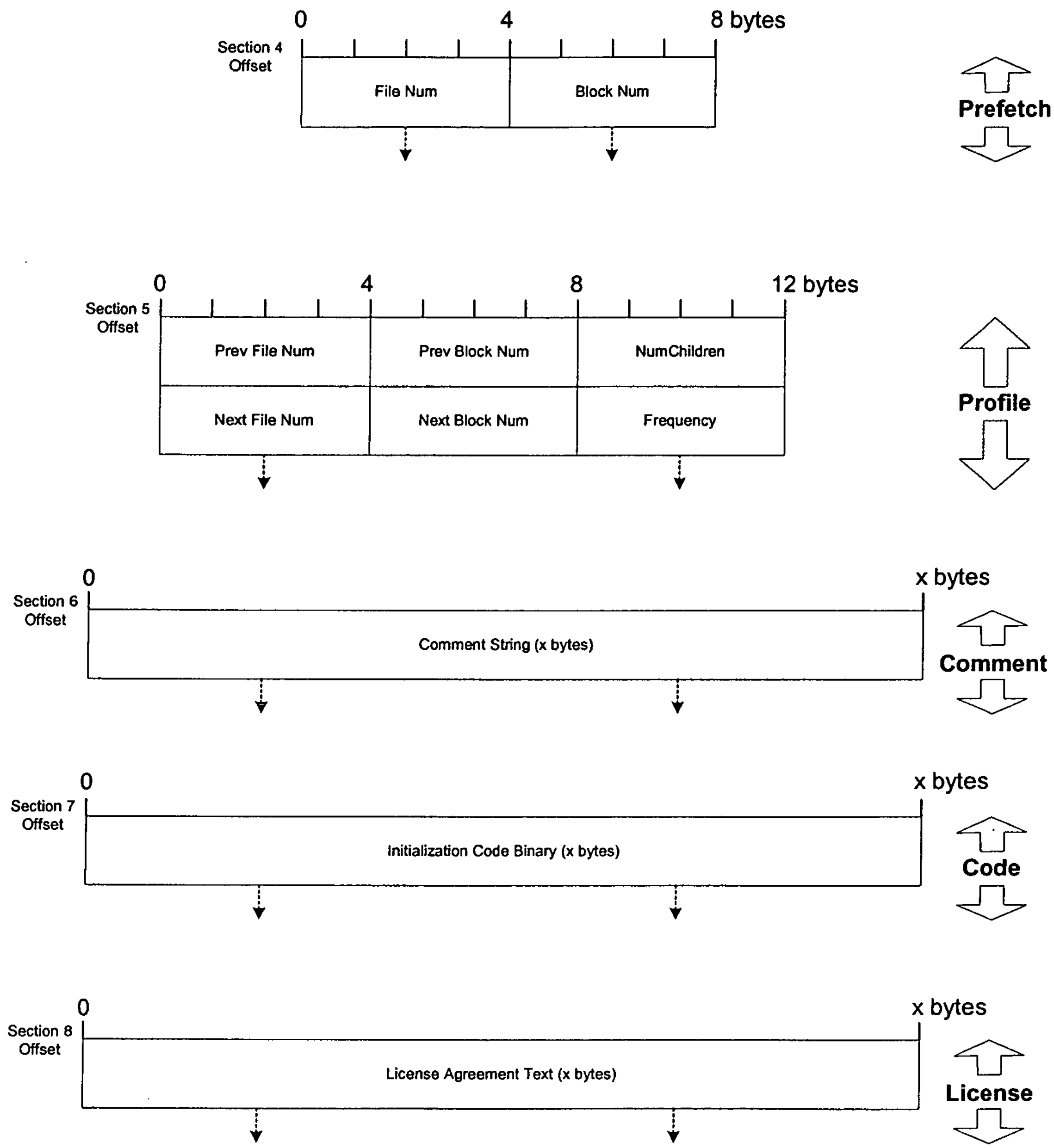
- What is the size of the AppInstallBlock for a typical application like Office?
- How large should the prefetch sections be for optimal run of an application? At minimum, it should contain at least start/termination code.
- How should the AppInstallBlock handle application license agreement text string? Add a new section or use comment section. Does the dialog need to have exactly the same interface as the license agreement dialog on the local installation?
- Currently, file section stores complete pathname including the drive letter. The installation may place files according to some variables like %System-

Root% or %UserProfile%. How does the Builder detect this so it can propagate this information to the client?

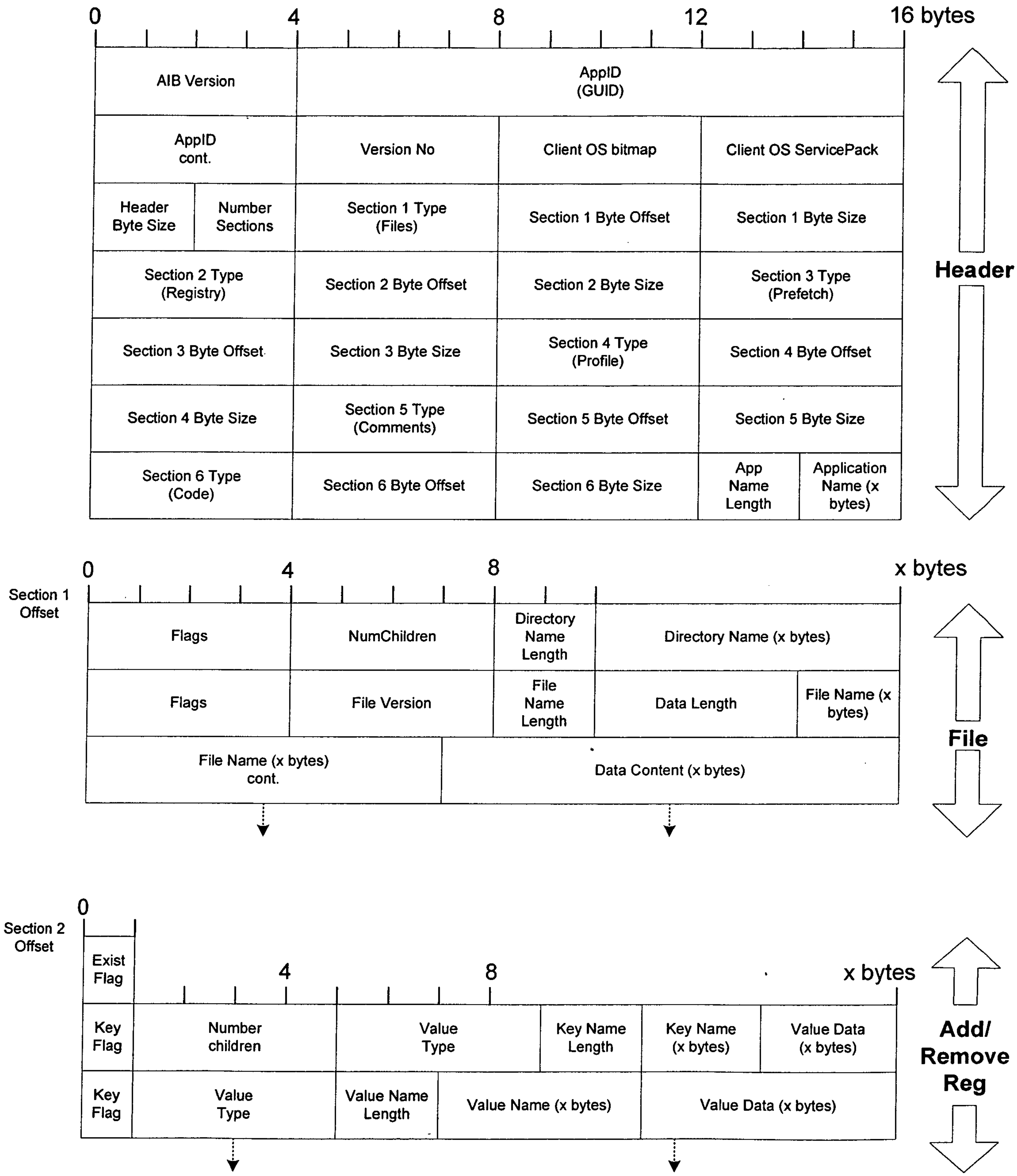
Format of AppInstallBlock (part 1 of 2)



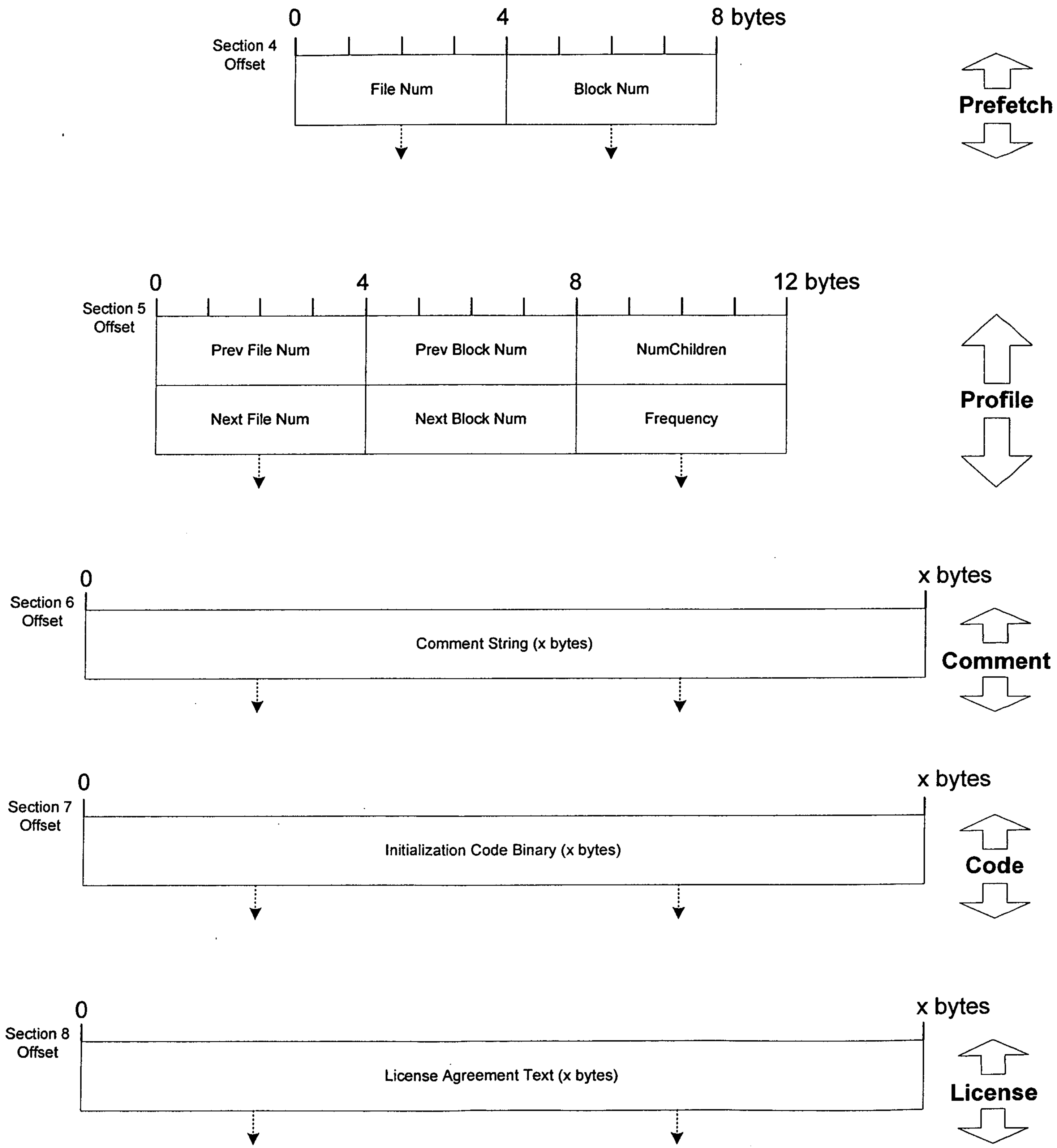
Format of AppInstallBlock (part 2 of 2)



Format of AppInstallBlock (part 1 of 2)



Format of AppInstallBlock (part 2 of 2)



Converting Apps for Stream Delivery and subsequent Execution

The Streamed Application Set Builder is a software program. It is used to convert locally installable applications into a data set suitable for streaming over the network. The streaming-enabled data set is called the Streamed Application Set (SAS). This document describes the procedure used to convert locally installable applications into the SAS.

The application conversion procedure into the SAS consists of the several phases. In the first phase, the Builder program monitors the installation process of a local installation of the desired application for conversion. The Builder monitors any changes to the system and records those changes in an intermediate data structure. After the application is installed locally, the Builder enters the second phase of the conversion. In the second phase, the Builder program invokes the installed application executable and obtains sequences of frequently accessed file blocks of this application. Both the Builder program and the client software use the sequence data to optimize the performance of the streaming process. Once the sequencing information is obtained, the Builder enters the final phase of the conversion. In this phase, the Builder gathers all data obtained from the first two phase and processes the data into the Streamed Application Set.

In the next sections, detailed descriptions of the three phases of the Builder conversion process are described. The three phases consists of installation monitoring (IM), application profiling (AP), and finally SAS packaging (SP). In most cases, the conversion process is general and applicable to all type of system. In places where the conversion is OS dependent, the discussion is focused on Microsoft Windows environment. Issues on conversion procedure for other OS environment are described in later sections.

Installation Monitoring (IM)

In the first phase of the conversion process, the Builder Installation Monitor (IM) component invokes the application installation program that installs the application locally. The IM observes all changes to the local computer during the installation. The changes may involve one or more of the following: changes to system or environment variables; and modifications, addition, or deletion of one or more files. The IM records all changes to the variables and files in a data structure to be sent to the Builder's Streamed Application Packaging component. In the following paragraphs, detailed description of the Installation Monitor is described for Microsoft Windows environment.

In Microsoft Windows system, the Installation Monitor (IM) component consists of a kernel-mode driver subcomponent and a user-mode subcomponent. The kernel-mode driver is hooked into the system registry and file system function interface calls. The hook into the registry function calls allows the IM to monitor system variable changes. The hook into the file system function calls enables the IM to observe file changes.

Installation Monitor Kernel-Mode subcomponent (IM-KM)
The IM-KM subcomponent monitors two classes of information during an application installation: system registry modifications and file modifications. Different techniques are used for each of these classes.

To monitor system registry modifications, the IM-KM component replaces all kernel-mode API calls in the System Service Table that write to the system registry with new functions defined in the IM-KM subcomponent. When an installation program calls one of the API functions to write to the registry, the IM-KM function is called instead, which logs the modification data (including registry key path, value name and value data) and then forwards the call to the actual operating system defined function. The modification data is made available to the IM-UM subcomponent through a mechanism described below in **Installation Monitor User-Mode subcomponent (IM-UM).**

To monitor file modifications, a *filter driver* is attached to the file system's driver stack. Each time an installation program modifies a file on the system, a function is called in the IM-KM subcomponent, which logs the modification data (including file path and name) and makes it available to the IM-UM using a mechanism described below in **Installation Monitor User-Mode subcomponent (IM-UM).**

The mechanisms used for monitoring registry modifications and file modifications will capture modifications made by any of the processes currently active on the computer system. While the installation program is running, other processes that, for example, operate the desktop and service network connections may be running and may also modify files or registry data during the installation. This data must be removed from the modification data to avoid inclusion of modifications that are not part of the application installation. The IM-KM uses process monitoring to perform this filtering.

To do process monitoring, the IM-KM installs a process notification callback function that is called each time a process is created or destroyed by the operating system. Using this callback function, the operating system sends the **created** process ID as well as the process ID of the **creator** (or parent) process. The IM-KM uses this information, along with the process ID of the IM-UM, to create a list of all of the processes created during the application installation. The IM-KM uses the following algorithm to create this list:

1. Before the installation program is launched by the IM-UM, the IM-UM passes its own process ID to the IM-KM. Since the IM-UM is launching the installation application, the IM-UM will be the ancestor (parent, grandparent etc) of any process (with one exception – the Installer Service described below) that modifies files or registry data as part of the application installation.
2. When the installation is launched and begins the creating processes, the IM-KM process monitoring logic is notified by the operating system via the process notification callback function.
3. If the creator (parent) process ID sent to the process notification callback function is already in the process list, the new process is included in the list.

When an application on the system modifies either the registry or files, and the IM-KM monitoring logic captures the modification data, but before making it available to the IM-

UM, it first checks to see if the process that modified the registry or file is part of the process list. It is only made available to the IM-UM if it is in the process list.

It is possible that a process that is not a process ancestor of the IM-UM will make changes to the system as a proxy for the installation application. Using interprocess communication, an installation program may request than an *Installer Service* make changes to the machine. In order for the IM-KM to capture changes made by the Installer Service, the process monitoring logic includes a simple rule that also includes any registry or file changes that have been made by a process with the same name as the Installer Service process. On Windows 2000, for example, the Installer Service is called "msi.exe".

Installation Monitor User-Mode subcomponent (IM-UM)
The IM kernel-mode (IM-KM) driver subcomponent is controlled by the user-mode subcomponent (IM-UM). The IM-UM sends messages to the IM-KM to start and stop the monitoring process via standard I/O control messages known as IOCTLs. The message that starts the IM-KM also passes in the process ID of the IM-UM to facilitate process monitoring described in the IM-KM description.

When the installation program modifies the computer system, the IM-KM signals a named kernel event. The IM-UM listens for these events during the installation. When one of these events is signaled, the IM-UM calls the IM-KM using an IOCTL message. In response, the IM-KM packages data describing the modification and sends it to the IM-UM.

The IM-UM sorts this data and removes duplicates. Also, it parameterizes all local-system-specific registry keys, value names, and values. For example, an application will often store paths in the registry that allow it to find certain files at run-time. These path specification must be replaced with parameters that can be recognized by the client installation software.

A user interface is provided for the IM-UM that allows an operator of the Builder to browse through the changes made to the machine and to edit the modification data before the data is packaged into an SAS.

Once the installation of an application completed, the IM-UM forwards data structures representing the file and registry modifications to the Streamed Application Packager. See figure 1 for the control flow diagram of IM module.

Monitoring Application Configuration
Using the techniques described above for monitoring file modifications and monitoring registry modifications, the builder can also monitor a running application that is being configured for a particular working environment. The data acquired by the IM-UM can be used to duplicate the same configuration on multiple machines, making it unnecessary for each user to configure his/her own application installation.

An example of this is a client server application for which the client will be streamed to the client computer system. Common configuration modifications can be captured by the IM and packed into the SAS. When the application is streamed to the client machine, it is already configured to attach to the server and begin operation.

Application Profiling (AP)
In the second phase of the conversion process, the Builder's Application Profiler (AP) component invokes the application executable program that is installed during the first phase of the conversion process. Given a particular user input, the executable program file blocks are accessed in a particular sequence. And the purpose of the AP is to captures this sequence data associated with some user inputs. This data is useful in several ways.

First of all, frequently used file blocks can be streamed to the client machine before other less used file blocks. A frequently used file blocks is cached locally on the client cache before the user starts using the streamed application for the first time. This has the effect of making the streamed application as responsive to the user as the locally installed application by hiding any long network latency and bandwidth problems.

Secondly, the frequently accessed files can be reordered in the directory to allow faster lookup of the file information. This optimization is useful for directories with large number of files. When the client machine looks up a frequently used file in a directory, it finds this file early in the directory search. In an application run with many directory queries, the performance gain is significant.

Finally, the association of a set of file blocks with a particular user input allows the client machine to request minimum amount of data needed to respond to that particular user command. The profile data association with a user command is sent from the server to the client machine in the AppInstallBlock during the 'preparation' of the client machine for streaming. When the user on a client machine invokes a particular command, the codes corresponding to this command is prefetched from the server.

The Application Profiler (AP) is not as tied to the system as the Installation Monitor (IM) but there are still some OS dependent issues. In the Windows system, the AP still has two subcomponents: kernel-mode (AP-KM) subcomponent and the user-mode (AP-UM) subcomponent. The AP-UM invokes the converting application executable. Then AP-UM starts the AP-KM to track the sequences of file block accesses by the application. Finally when the application exits after the pre-specified amount of sequence data is gathered, the AP-UM retrieves the data from AP-KM and forwards the data to the Streamed Application Packager. See Figure 2 for the control flow diagram of the AP module.

Streamed Application Set Packaging (SP)
In the final phase of the conversion process, the Builder's Streamed Application Set Packager (SP) component processes the data structure from IM and AP to create a data set suitable for streaming over the network. This converted data set is called the

Streamed Application Set and is suitable for uploading to the Streamed Application Servers for subsequent downloading by the stream client. Figure 3 shows the control flow of the SP module.

Each file included in a Streamed Application Set is assigned a file number that identifies it within the SAS.

The Streamed Application Set consists of the three sets of data from the Streamed Application Server's perspective. The three types of data are the Concatenation Application File (CAF), the Size Offset File Table (SOFT), and the Root Versioning Table (RVT).

The Concatenation Application File (CAF) consists of all the files and directories needed to stream to the client. The CAF can be further divided into two subsets: initialization data set and the runtime data set.

The initialization data set is the first set of data to be streamed from the server to the client. This data set contains the information captured by IM and AP needed by the client to prepare the client machine for streaming this particular application. This initialization data set is also called the AppInstallBlock (AIB). In addition to the data captured by the IM and AP modules, the SP is also responsible for merging any new dynamic profile data gathered from the client and the server. This data is merged into the existing AppInstallBlock to optimize subsequent streaming of the application. With the list of files obtained by the IM during application installation, the SP module separates the list of files into regular streamed files and the spoof files. The spoof files consists of those files not installed into standard application directory. This includes files installed into system directories and user specific directories. The detailed format description of the AppInstallBlock is described in Appendix B.

The second part of the CAF consists of the runtime data set. This is the rest of the data that is streamed to the client once the client machine is initialized for this particular application. The runtime data consists of all the regular application files and the directories containing information about those application files. Detailed format description of the runtime data in the CAF section is described in Appendix A. The SP appends every files recorded by IM into the CAF and generates all directories. Each directory contains list of file name, file number, and the metadata associated with the files in that particular directory.

The SP is also responsible for generating the SOFT file. This is a table used to index into the CAF for determining the start and the end of a file. The server uses this information to quickly access the proper file within the directory for serving the proper file blocks to the client.

Finally, the SP creates the RVT file. The Root Versioning Table contains a list of root file number and version number. This information is used to track minor application patches and upgrades. Each entry in the RVT corresponds to one patch level of the

application with a corresponding new root directory. The SP generates new parent directories when any single file in that subdirectory tree is changed from the patched upgrade. The RVT is uploaded to the server and requested by the client at appropriate time for the most updated version of the application by a simple comparison of the client's Streamed Application root file number with the RVT table located on the server once the client is granted access authorization to retrieve the data. Figure 4 shows the internal representation of a simple SAS before and after a new file is added to a new version of an application.

Data Flow Description

The following list describes the data that is passed from one component to another. The numbers corresponds to the numbering in the Data Flow diagram.

Install Monitor

1. The full pathname of the installer program is query from the user of the Builder program and is sent to the Install Monitor.
2. The Install Monitor (IM) user-mode sends a read request to the OS to spawn a new process for installing the application on the local machine.
3. The OS loads the application installer program into memory and run the application installer program. OS returns the process ID to the IM.
4. The application program is started by the IM-UM.
5. The application installer program sends read request to the OS to read the content of the CD.
6. The CD media data files are read from the CD.
7. The files are written to the appropriate location on the local hard-drive.
8. IM kernel-mode captures all file read/write requests and all registry read/write requests by the application installer program.
9. IM user-mode program starts the IM kernel-mode program and sends the request to start capturing all relevant file and registry data.
10. IM kernel-mode program sends the list of all file modifications, additions, and deletions; and all registry modifications, additions, and deletions to the IM user-mode program.
11. Inform the SAS Builder UI that the installation monitoring has completed and display the file and registry data in a graphical user interface.

Application Profiler

12. Builder UI invokes Application Profiling (AP) user-mode program by querying the user for the list of application executable names to be profiled. The AP user-mode also query the user for division of file blocks into sections corresponding to the commands invoked by the user of the application being profiled.
13. Application Profiler user-mode invokes each application executable in succession by spawning each program in a new process. The OS loads the application executable into memory, run the application executable, and return the process ID to the Application Profiler.

14. During execution, the OS on behalf of the application send the request to the hard-drive controller to read the appropriate file blocks into memory as needed by the application.
15. The hard-drive controller returns all file blocks requested by the OS.
16. Every file accesses to load the application file blocks into memory are monitored by the Application Profiler (AP) kernel-mode program.
17. The AP user-mode program informs the AP kernel-mode program to start monitoring relevant file accesses.
18. Application Profiler kernel-mode returns the file access sequence and frequency information to the user-mode program.
19. Application Profiler returns the processed profile information. This has two sections. The first section is used to identify frequency of files accessed. The second section is used to list the file blocks for prefetch to the client. The file blocks can be further categorized into subsections according to the commands invoked by the user of the application.

SAS Packager

20. The Streamed Application Packager receives files and registry changes from the Builder UI. It also receives the file access frequency and a list of file blocks from the Profiler. File numbers are assigned to each file.
21. The Streamed Application Packager reads all file data from the hard-drive that are copied there by the application installer.
22. The Streamed Application Packager also reads the previous version of Streamed Application Set for support of minor patch upgrades.
23. Finally, the new Streamed Application Set data is stored back to non-volatile storage.
24. For new profile data gathered after the SAS has been created, the packager is invoked to update the AppInstallBlock in the SAS with the new profile information.

Mapping of Data Flow to Streamed Application Set (SAS)

Step 7: Data gathered from this step consist of the registry and file modification, addition, and deletion. This data is mapped to the AppInstallBlock's File Section, Add Registry Section, and Remove Registry Section.

Step 8 & 19: File data are copied to the local hard-drive then concatenated into part of the CAF contents. Part of the data is identified as spoof or copied files and the file names and/or contents are added to the AppInstallBlock.

Step 15 & 21: Part of the data gathered by the Profiler or gathered dynamically by the client is used in the AppInstallBlock as a prefetch hint to the client. Another part of the data is used to generate a more efficient SAS Directory content by ordering the files according the usage frequency.

Step 20: If the installation program was an upgrade, SAS Packager needs previous version of the Streamed Application Set data. Appropriate data from the previous version is combined with the new data to form the new Streamed Application Set.

Format of Streamed Application Set

The format of the Streamed Application Set consists of 3 sections: Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF). The RVT section lists all versions of the root file numbers available in a Streamed Application Set. The SOFT section consists of the pointers into the CAF section for every file in the CAF. The CAF section contains the concatenation of all the files associated with the streamed application. The CAF section is made up of regular application files, SAS directory files, AppInstallBlock, and icon files. Please see Appendix A for detailed information on the content of the SAS file. Figure 6 shows a typical layout of the SAS file.

OS dependent format

The format of the Streamed Application Set is designed to be as portable as possible across all OS platforms. At the highest level, the format of CAF, SOFT, and RVT that make up the format of Streamed Application Set are completely portable across any OS platforms. One piece of data structure that is OS dependent is located in the initialization data set called AppInstallBlock in the CAF. This data is dependent on the type of OS due to the differences in low-level system differences among different OS. For example, the Microsoft Windows contain system environment variables called the Registry. The Registry has a particular tree format not found in other operating systems like UNIX or MacOS.

Another OS dependent piece of data is located in the SAS directory files in the CAF. The directory contains file metadata information specific to Windows files. For example on the UNIX platform, there does not exist a hidden flag. This platform specific information needs to be transmitted to the client to fool the streamed application into believing that the application data is located natively on the client machine with all the associated file metadata in tack. If SAS is to be used to support streaming of UNIX or MacOS applications, file metadata specific to those systems will need to be recorded in the SAS directory.

Lastly, the format of the file names itself is OS dependent. Applications running on the Windows environment inherit the old MSDOS 8.3 file name format. To support this properly, the format of the SAS Directory file in CAF requires an additional 8.3 field to store this information. This field is not needed in other operating systems like UNIX or MacOS.

Device driver versus file system paradigm

The SAS client Prototype is implemented using the 'device driver' paradigm. One of the advantages of the device driver approach is that the caching of the sector blocks is simpler. In the device driver approach, the client cache manager only needs to track sector number in its cache. In comparison with the 'file system' paradigm, more complex data structure is required to track a subset of a file that is cached on a client machine. This makes 'device driver' paradigm easier to implement.

On the other hand, there are many drawbacks to the 'device driver' paradigm. On the Windows system, the device driver approach has problem supporting large number of

applications. This is due to the limitation on the number of assignable drive letters available in a Windows system (26 letters); and the fact that each application needs to be located on its own device. Note that having multiple applications on a device is possible, but then the server needs to maintain exponential number of devices that support all possible combinations of applications. This is too costly to maintain on the server.

Another problem with the device driver approach is that the device driver operates at the disk sector level. This is a much lower level than operating at the file level in the file system approach. The device driver does not know anything about files. Thus, the device driver cannot easily interact with the file level issues. For example, spoofing files and interacting with OS buffer cache is nearly impossible with device driver approach. But both spoofing files and interacting with OS buffer cache is need to get higher performance.

See figure 7 and 8 for the differences between two paradigms.

Implementation in the Prototype
The prototype has been implemented and tested successfully on the Windows and Linux distributed system. The prototype is implemented using the 'device driver' paradigm as described above. The exact procedure for streaming application data is described next.

First of all, the prototype server is started on either the Windows-based or Linux-based system. The server creates a large local file mimicking large local disk images. Once the disk images are prepared, it listens to TCP/IP ports for any disk sector read or write requests.

Secondly, the conversion process is done on a Windows system via semi-manual procedure. The server disk image is 'mounted' on the local Z drive by making the proper TCP/IP connection to the server. Then the application installation program is invoked and the application is installed into the Z drive. This writes the application files into the Z drive device driver, through the TCP/IP connection, and finally on to the server disk image. At the same time, a file and registry monitoring program records all registry and file changes. This data is stored as an initialization file to be invoked on the client to prepare the client machine for streaming.

Finally, after the application files is stored on the server disk image, the client prototype is started. The client connects to the server and 'mount' the server disk image as a local Z drive. Then the initialization file is invoked which setup the local registry variables and copy system files into proper directories. Once the local machine is prepared for streaming that particular application, the user can start using the application. When the application is first started, the pages are not located in the local buffer cache. The OS makes sector request to the Streamed Application device driver that forwards the sector request to the Streamed Application Cache Manager. If the sector is located in the Streamed Application cache, then the data is returned immediately. If the data is not located in the Streamed Application cache, then the request forwarded to the network component that sends the message to the server. The server finds the proper sector data

and returns the data to the client. The client Streamed Application Cache Manager caches the new sector data and forwards the sector data to the Streamed Application device driver. The device driver returns the sector data to the OS.

Implementation of SAS Builder

The SAS Builder has been implemented on the Windows-based platform. A preliminary Streamed Application Set file has been created for real-world applications like Adobe Photoshop. A simple extractor program has been developed to extract the SAS data on a pristine machine without the application installed locally. Once the extractor program is run on the SAS, the application runs as if it was installed locally on that machine. This process verifies the correctness of the SAS Building process.

Appendix A: Format of Streamed Application Set (SAS)

Functionality

The streamed application set is a data set associated with an application suitable for streaming over the network. The SAS is generated by the SAS Builder program. This program converts locally installable applications into the SAS. This document describes the format of the SAS.

Note: Fields greater than a single byte is stored in little-endian format. The Stream Application Set (SAS) file size is limited to 2^64 bytes. The files in the CAF section are laid out in the same order as its corresponding entries in the SOFT table.

Data type definitions

The format of the SAS consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the SAS.
- **ESSVersion [4 bytes]:** Version number of the SAS file format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Window Guidgen API is used to create this identifier.
- **Flags [4 bytes]:** Flags pertaining to SAS.
- **Reserved [32 bytes]:** Reserved spaces for future.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VendorNameLength [4 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.
- **AppBaseNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **AppBaseNameLength [4 bytes]:** Byte length of the application base name.

- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.
- **MessageIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **MessageLength [4 bytes]:** Byte length of the message text.
- **Message [X bytes]:** Message text. Null-terminated.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each SAS in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the SAS Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this SAS. The number indicates the number of entries in the Root Version Table (RVT).
-

Root Version structure: (variable number of entries)
- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VersionNameLength [4 bytes]:** Byte length of the version name
- **VersionName [X bytes]:** Application version name. I.e. "SP 1".
- **Metadata [32 bytes]:** See SAS FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1. The start of the SOFT table is aligned to 8 bytes boundary for faster access.

SOFT entry structure: (variable number of entries)
- **Offset [8 bytes]:** Byte offset into CAF of the start of this file.
- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an SAS FS directory file, or an icon file.

a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

b. *AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see Appendix B.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data.
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. *SAS Directory*

An SAS Directory contains information about the subdirectories and files located within this directory. This information is used to store metadata information related to the files associated with the streamed application. This data is used to fool application into thinking that it is running locally on a machine when most of the data is resided elsewhere.

The SAS directory contains information about files in its directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for SAS directory file.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.
- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.

Variable-Sized File Entry:

- **UsedFlag [1 byte]:** 1 for used, 0 for unused.
- **ShortLen [1 byte]:** Length of short file name.
- **LongLen [2 byte]:** Length of long file name.
- **NameHash [4 bytes]:** Hash value of the short file name for quick lookup without comparing whole string.
- **ShortName [24 bytes]:** 8.3 short file name in Unicode. Not null-terminated.
- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **SAS flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
    - **Bit 0:** Read-only – Set if file is read-only
    - **Bit 1:** Hidden – Set if file is hidden from user
    - **Bit 2:** Directory – Set if the file is an SAS Directory
    - **Bit 3:** Archive – Set if the file is an archive
    - **Bit 4:** Normal – Set if the file is normal
    - **Bit 5:** System – Set if the file is a system file
    - **Bit 6:** Temporary – Set if the file is temporary

  The bits of the **SAS flags** have the following meaning:
    - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
    - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
    - **Bit 2:** Read-only – Set if the file is read-only
- **LongName [X bytes]:** Long filename in Unicode format with null-termination character.

d. Icon files

- **IconFileData [X bytes]:** Content of an icon file.

Appendix B: Format of AppInstallBlock

Functionality

The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the SAS client to 'initialize' the client machine before the streamed application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that streamed application.

The AppInstallBlock is created offline by the SAS Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system, and any files added or modified in the system directories. Files added to the application specific directory is not recorded in the AppInstallBlock to reduce the amount of time needed to send the AppInstallBlock to the SAS client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

The AppInstallBlock and the runtime data are packaged into the SAS by the Builder and then uploaded to the application server. After the SAS client subscribed to an application and before the application is run for the first time, the AppInstallBlock is send by the server to the client. The SAS client invokes the default initialization procedure and the optional application-specific initialization code. Together, the default and the application-specific initialization procedure process the data in the AppInstallBlock to make the machine ready for streaming that particular application.

Data type definitions

The AppInstallBlock is divided into the following sections: header section, variable section, file section, profile section, prefetch section, comment section, and code section. The header section contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section is a tree structure consisting of the files copied to C drive during the application installation. The profile section contains the initial set of block reference sequences during Builder profiling of the application. The prefetch section consists of a subset of profiled blocks used by the Builder as a hint to the SAS client to prefetch initially. The comment section is used to inform the SAS client user of any relevant

information about the application installation. Finally, the code section contains an optional program tailored for any application-specific installation not covered by the default streamed application installation procedure. In Windows version, the code section contains a Windows DLL. The following is a detailed description of each fields of the AppInstallBlock.

Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4K byte size.

1. Header Section:

The header section contains the basic information about this AppInstallBlock. This includes the versioning information, application identification, and index into other sections of the file.

Core Header Structure:

- **AibVersion [4 bytes]**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).
- **AppId [16 bytes]**: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.
- **VersionNo [4 bytes]**: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.
- **ClientOSBitMap [4 bytes]**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future OSs we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- **ClientOSServicePack [4 bytes]**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap.
- **Flags [4 bytes]**: Flags pertaining to AppInstallBlock
  - **Bit 0:** Reboot – If set, the SAS client needs to reboot the machine after installing the AppInstallBlock on the client machine.
  - **Bit 1:** Unicode – If set, the string characters are 2 bytes wide instead of 1 byte.
- **HeaderSize [2 bytes]**: Total size in bytes of the header section.
- **Reserved [32 bytes]**: Reserved spaces for future.
- **NumberOfSections [1 byte]**: Number of sections in the index table. This determines the number of entries in the index table structure described below:

Index Table Structure: (variable number of entries)

- **SectionType [1 bytes]**: The type of data describe in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.
- **SectionOffset [4 bytes]**: The offset from the beginning of the file indicates the beginning of section.
- **SectionSize [4 bytes]**: The size in bytes of section.

Variable Structure:

- **ApplicationNameIsAnsi [1 byte]**: 1 if ansi, 0 if Unicode.
- **ApplicationNameLength [4 bytes]**: Byte size of the application name
- **ApplicationName [X bytes]**: Null terminating name of the application

2. File Section:

The file section contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into 'unusual' directory during the installation of an application. If the file content is small, the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is list of trees stored in a contiguous sequence of address space according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directory. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

Directory Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 0 is set if this entry is a directory
- **NumberOfChildren [2 bytes]**: Number of nodes in this directory
- **DirectoryNameLength [4 bytes]**: Length of the directory name
- **DirectoryName [X bytes]**: Null terminating directory name

Leaf Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 1 is set to 1 if this entry is a spoof or copied file name
- **FileVersion [8 bytes]**: Version of the file GetFileVersionInfo() if the file is win32 file image. Need variable file version size returned by GetFileVersionInfoSize(). Otherwise use file size or file modified time to compare which file is the later version.
- **FileNameLength [4 bytes]**: Byte size of the file name
- **FileName [X bytes]**: Null terminating file name

- **DataLength [4 bytes]**: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file
- **Data [X bytes]**: Either the spoof file name or the content of the copied file

3. Add Variable and Remove Variable Sections:

The add and remove variable sections contain the system variable changes needed to run the application. In Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address space according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

a. Registry Subsection:

1. "HKCR": HKEY_CLASSES_ROOT
2. "HKCU": HKEY_CURRENT_USER
3. "HKLM": HKEY_LOCAL_MACHINE
4. "HKUS": HKEY_USERS
5. "HKCC": HKEY_CURRENT_CONFIG

Tree Structure: (5 entries)

- **ExistFlag [1 byte]**: Set to 1 if this tree exist, 0 otherwise.
- **Key or Value Structure entries [X bytes]**: Serialization of the tree into variable number key or value structures described below.

Key Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **NumberOfSubchild [4 bytes]**: Number of subkeys and values in this key directory
- **KeyNameLength [4 bytes]**: Byte size of the key name
- **KeyName [X bytes]**: Null terminating key name

Value Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **ValueType [4 byte]**: Type of values from the Win32 API function RegQueryValueEx(): REG_SZ, REG_BINARY, REG_DWORD, REG_LINK, REG_NONE, etc...
- **ValueNameLength [4 bytes]**: Byte size of the value name
- **ValueName [X bytes]**: Null terminating value name
- **ValueDataLength [4 bytes]**: Byte size of the value data

- **ValueData [X bytes]**: Value of the Data

In addition to registry changes, an installation in Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the SAS client machine. The ini entries are appended to the end of the variable section after the 5 registry trees are enumerated.

b. INI Subsection:

- **NumFiles [4 bytes]**: Number of INI files modified.

File Structure: (variable number of entries)

- **FileNameLength [4 bytes]**: Byte length of the file name
- **FileName [X bytes]**: Name of the INI file
- **NumSection [4 bytes]**: Number of sections with the changes

Section Structure: (variable number of entries)

- **SectionNameLength [4 bytes]**: Byte length of the section name
- **SectionName [X bytes]**: Section name of an INI file
- **NumValues [4 bytes]**: Number of values in this section

Value Structure: (variable number of entries)

- **ValueLength [4 bytes]**: Byte length of the value data
- **ValueData [X bytes]**: Content of the value data

4. Prefetch Section:

The prefetch section contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing for frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of blocks to include in the prefetch section is the blocks associated with frequently accessed directories and file metadata in this directory. The prefetch section is divided into two subsections. One part contains the critical blocks that are used during startup of the streamed application. The second part consists of the blocks accessed for common user operations like opening and saving of document. The format of the data is described below:

a. Critical Block Subsection:

- **NumCriticalBlocks [4 bytes]:** Number of critical blocks.

Block Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

b. Common Block Subsection:

- **NumCommonBlocks [4 bytes]:** Number of critical blocks.

Block Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

5. Profile Section:

The profile section consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry [*row, column*] of the matrix is the frequency a block *row* is followed by a block *column*. In any realistic applications of fair size, this matrix is very large and sparse. Proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the SAS client prefetcher performance can be increased.

Row Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the row block
- **BlockNumber [4 bytes]**: Block Number of the row block
- **NumberColumns [4 bytes]**: number of blocks that follows this block. This field determines the number of column structures following this field.

Column Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the column block
- **BlockNumber [4 bytes]**: Block Number of the column block
- **Frequency [4 bytes]**: frequency the row block is followed by column block

6. Comment Section:

The comment section is used by the Builder to describe this AppInstallBlock in more detail.

- **CommentLengthIsAnsi [1 byte]**: 1 if string is ansi, 0 if Unicode format.
- **CommentLength [4 bytes]**: Byte size of the comment string
- **Comment [X bytes]**: Null terminating comment string

7. Code Section:

The code section consists of the application-specific initialization code needed to run on the SAS client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the SAS client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: *Install()*, *Uninstall()*. The SAS client loads the DLL and invokes the appropriate function calls.

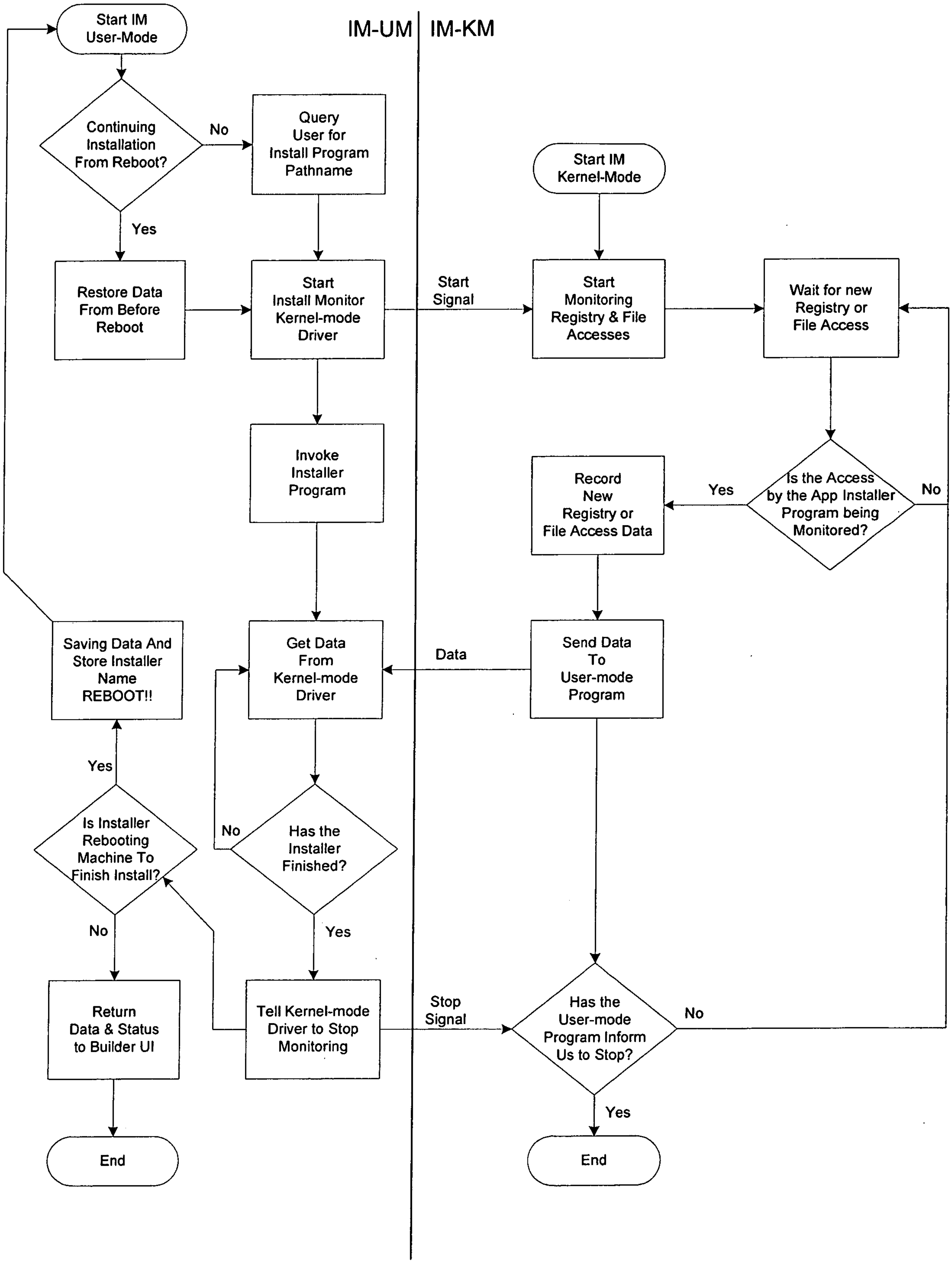
- **CodeLength [4 bytes]**: Byte size of the code
- **Code [X bytes]**: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

8. LicenseAgreement Section:

The Builder creates the license agreement section. The SAS client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

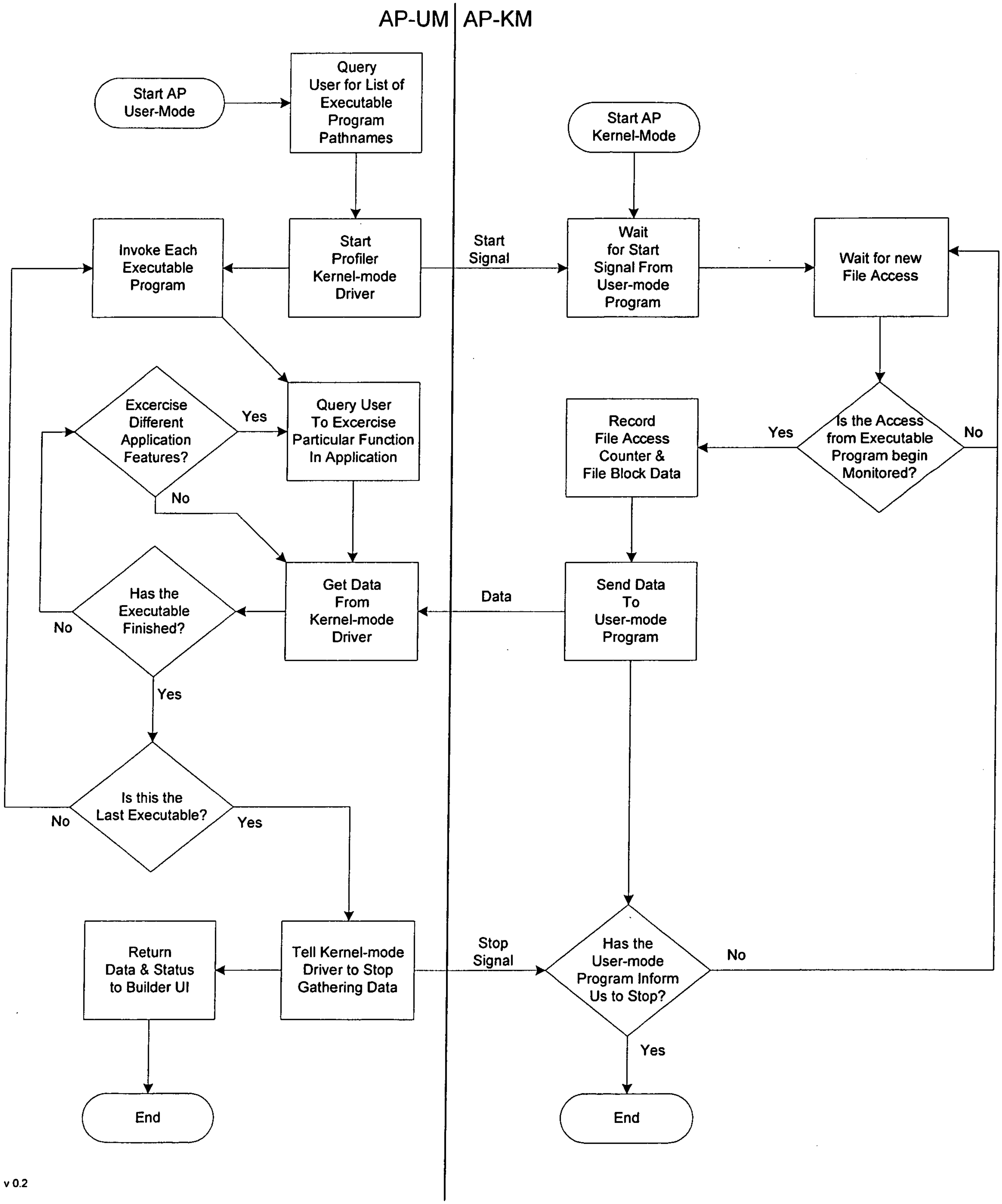
- **LicenseTextIsAnsi [1 byte]**: 1 if ansi, 0 if Unicode format.
- **LicenseTextLength [4 bytes]**: Byte size of the license text
- **LicenseAgreement [X bytes]**: Null terminating license agreement string

Figure 1: Builder Install Monitor (IM) Control Flow Diagram



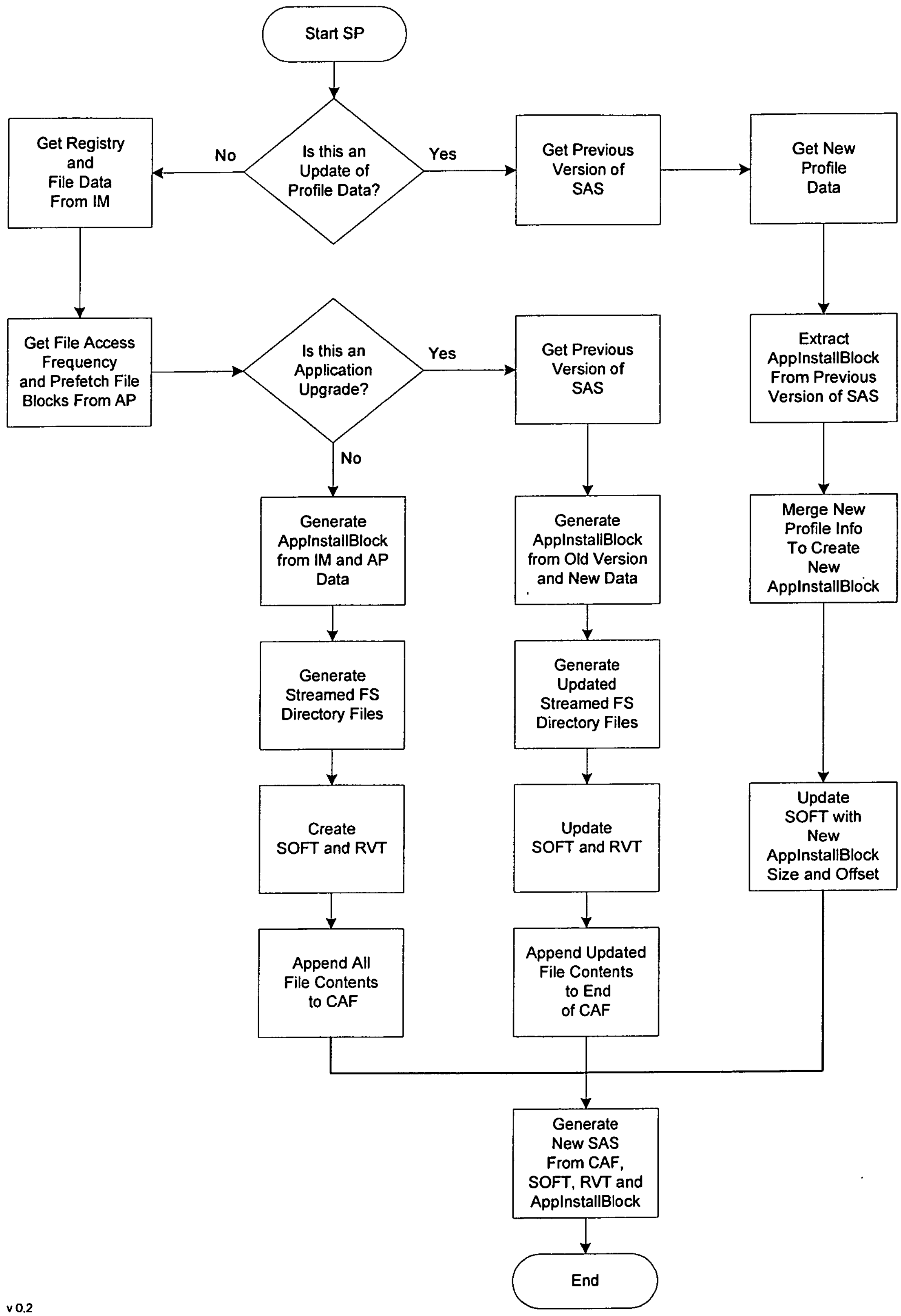
v 0.2

Figure 2: Builder Application Profiler (AP) Control Flow Diagram



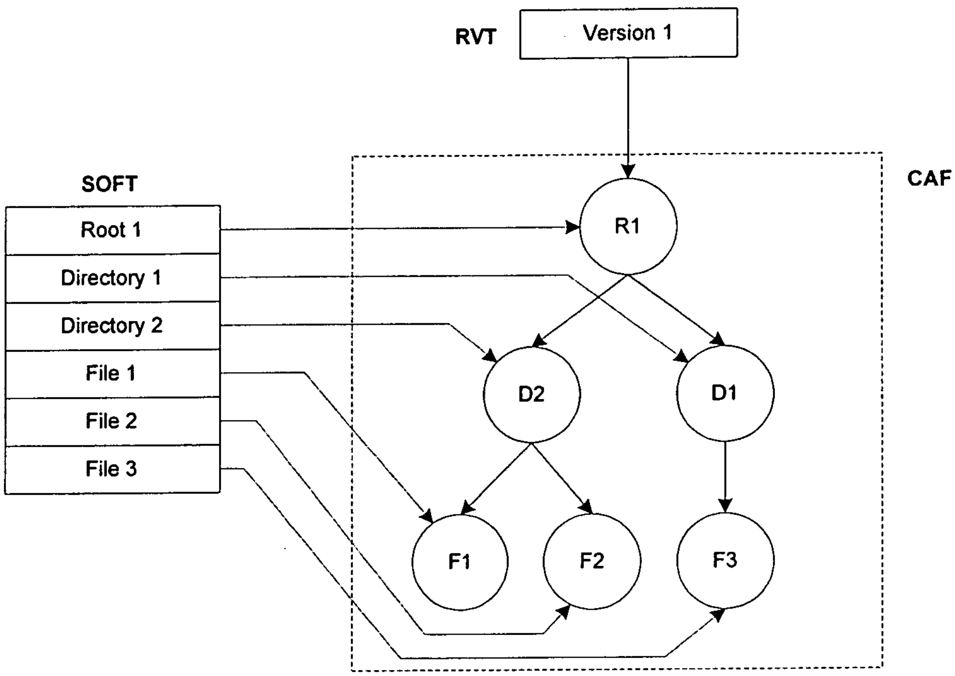
v 0.2

Figure 3: Builder SAS Packager (SP) Control Flow Diagram

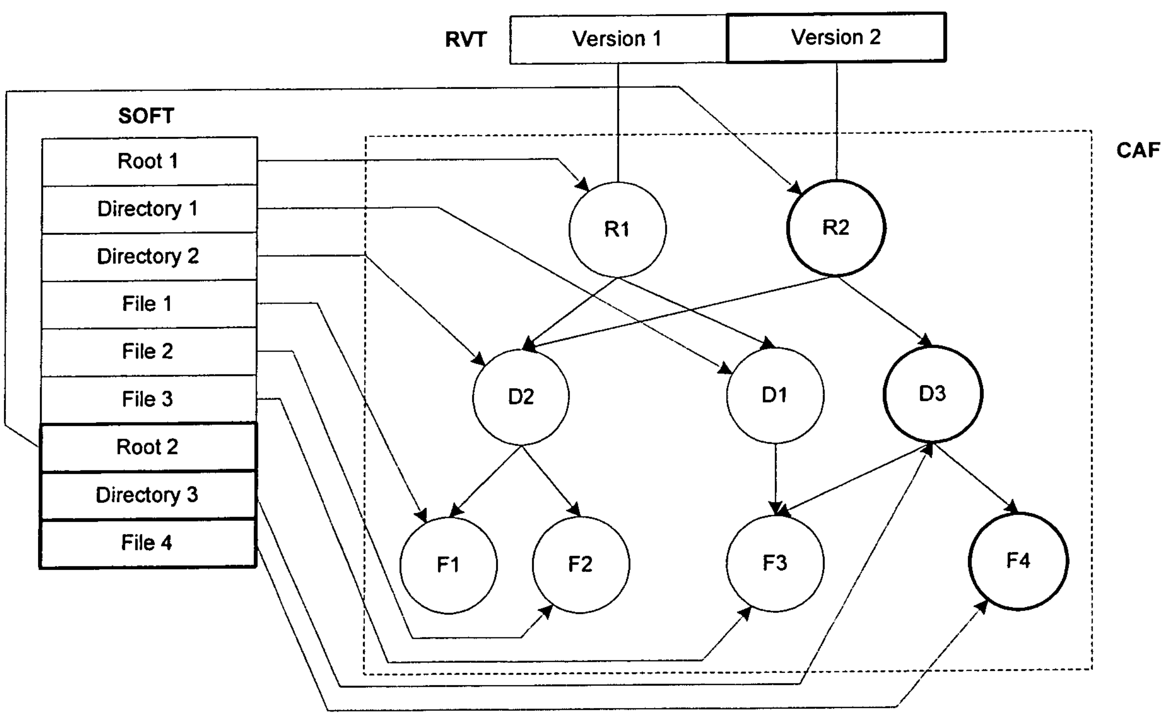


v 0.2

Figure 4: Internal Representation of SAS with Support for Versioning

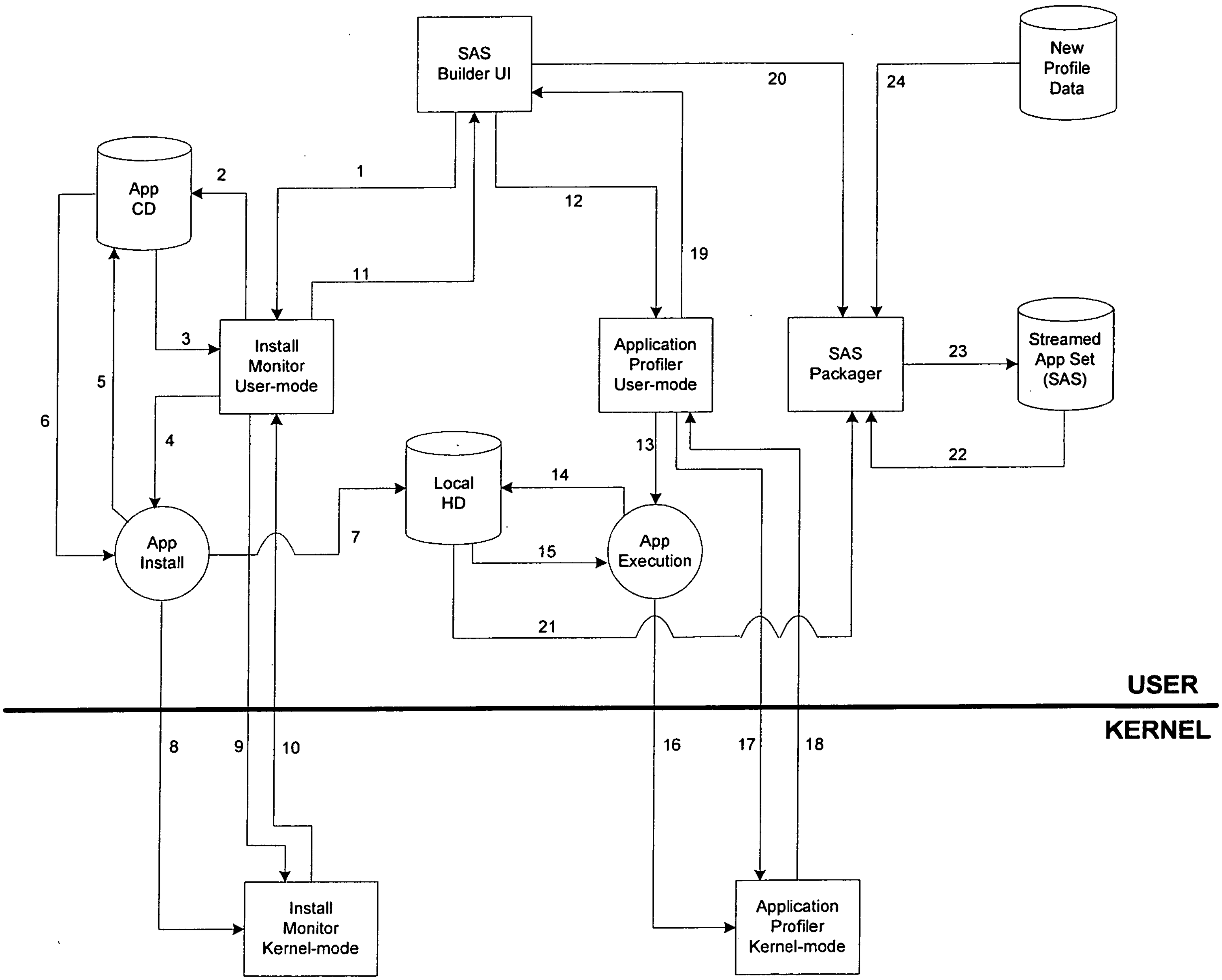


Version 1 of Application in SAS



Version 2 of Application in SAS with a New File

Figure 5: Streamed Application Set (SAS) Builder Data Flow Diagram



v 0.2

Figure 6: Content of a Streamed Application Set

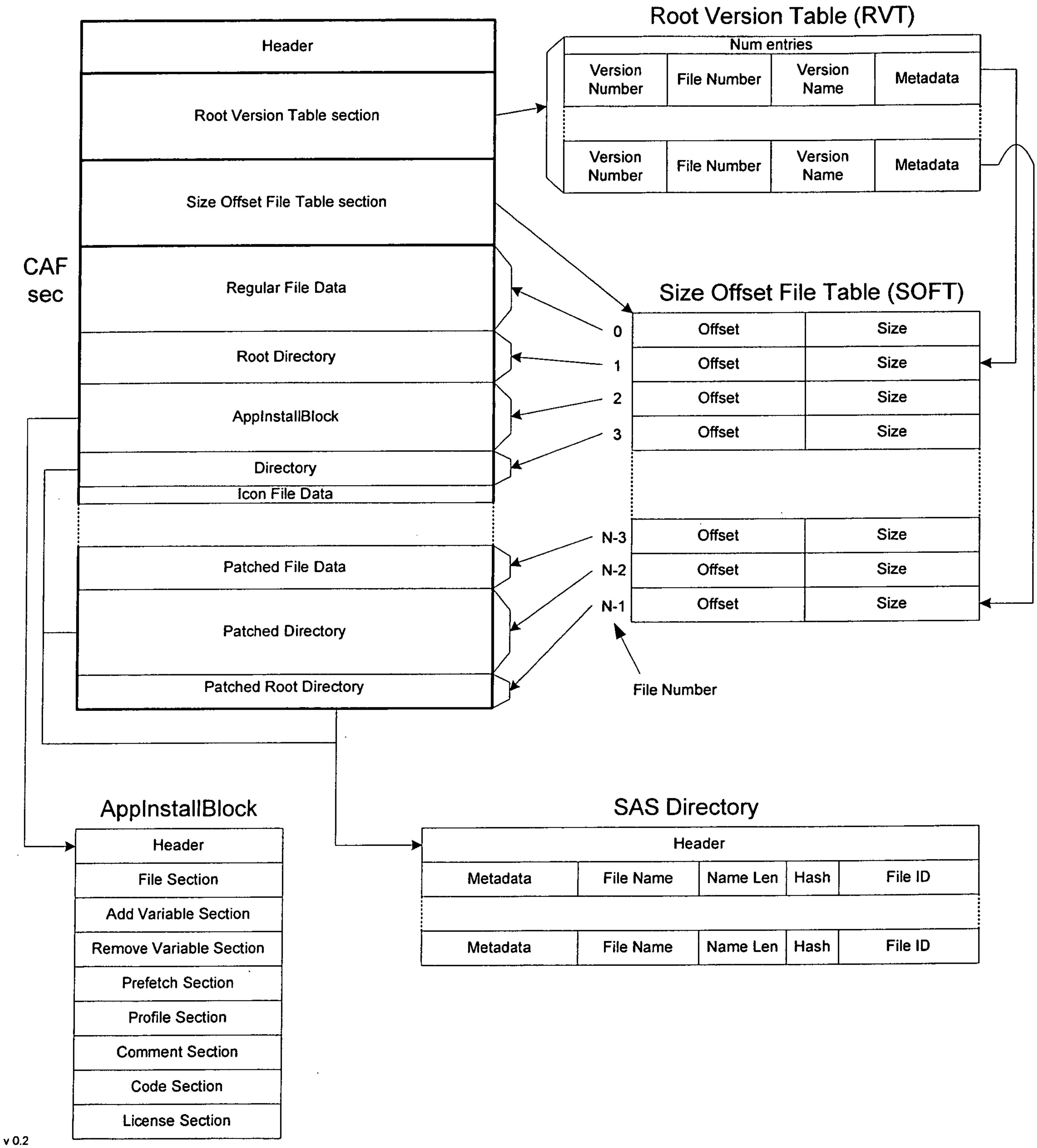


Figure 7: Device Driver Paradigm

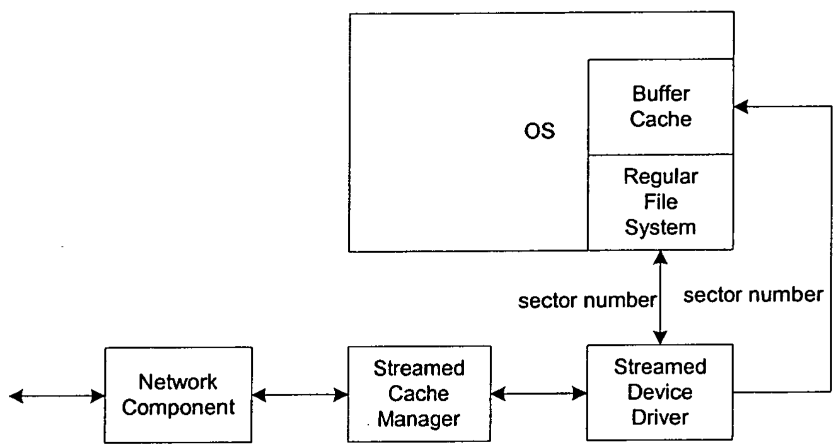


Figure 8: File System Paradigm

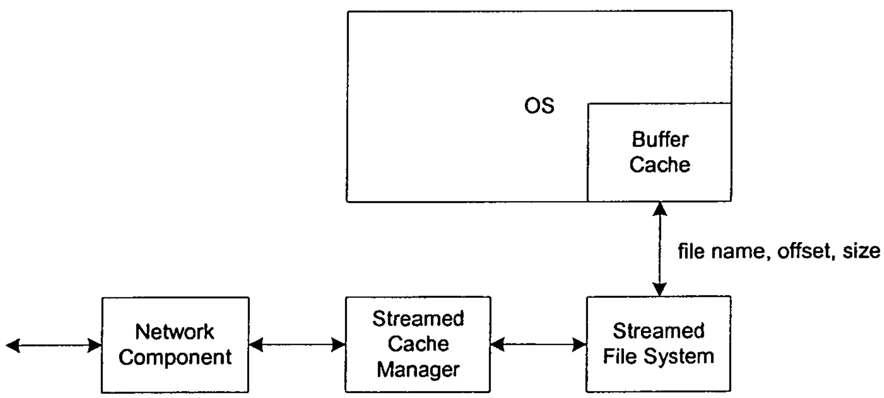
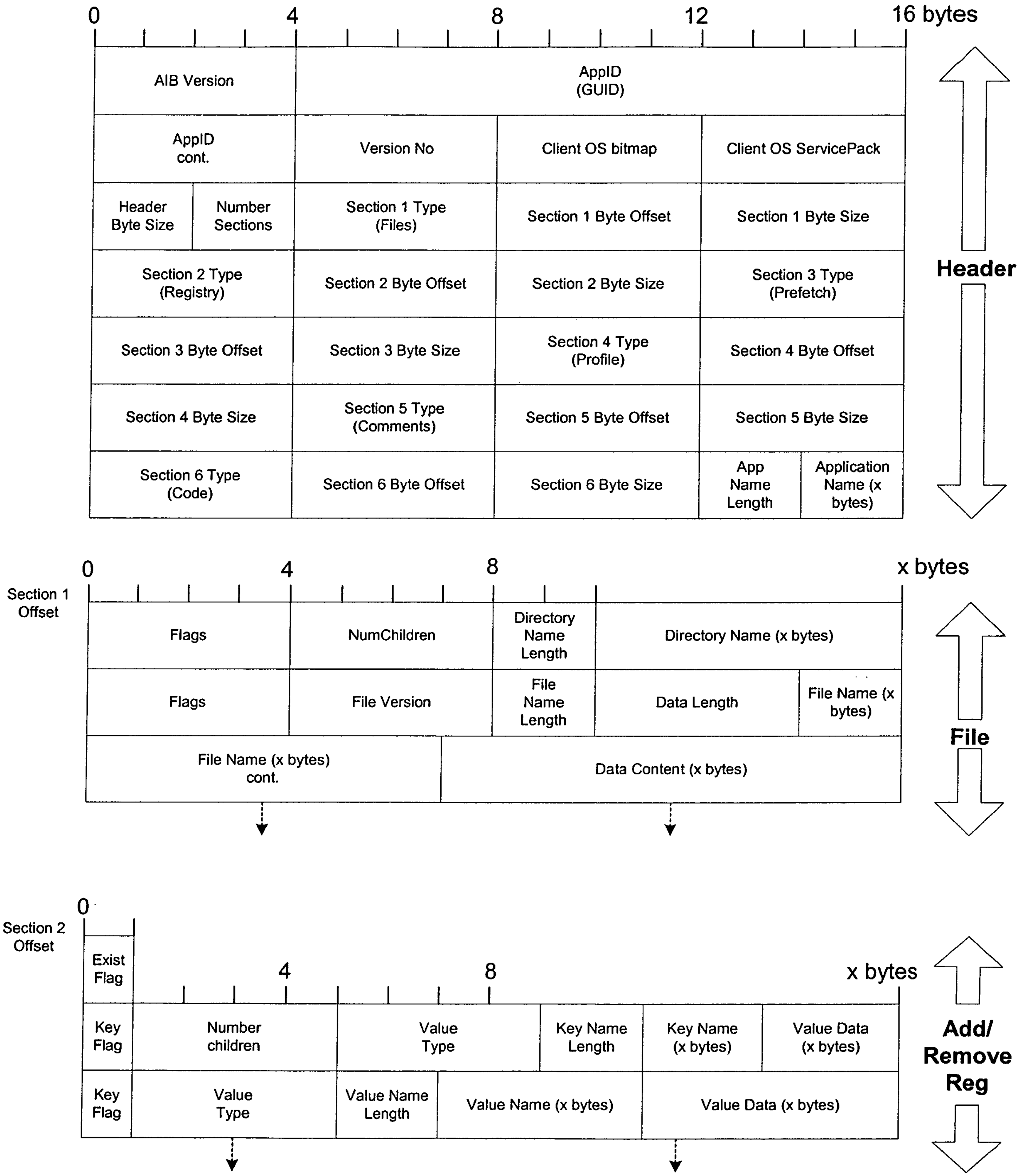
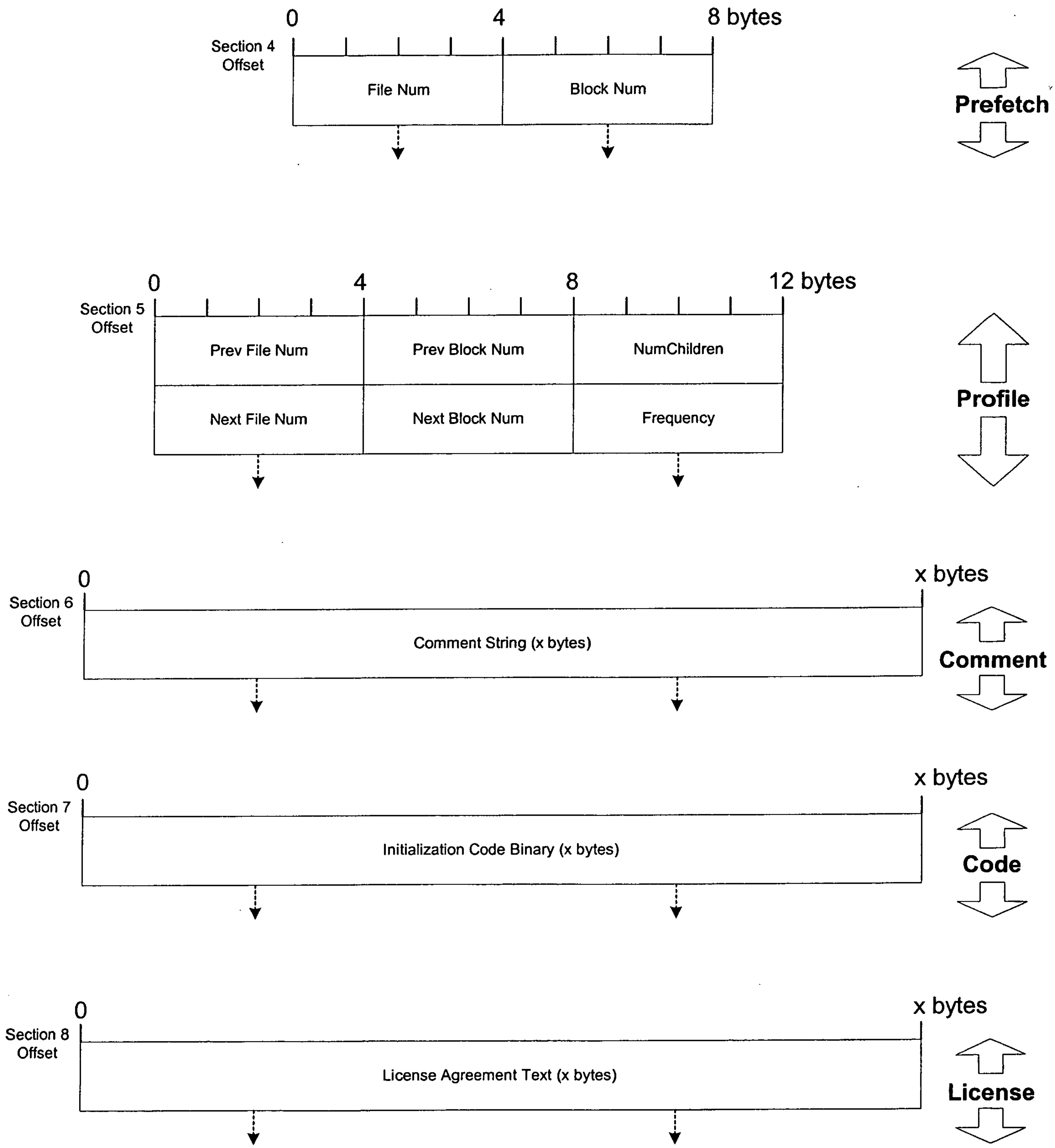


Figure 10: Format of AppInstallBlock (part 1 of 2)



Format of AppInstallBlock (part 2 of 2)



eStream Application Builder High-Level Design

Authors: Sanjay Pujare and David Lin

Version 0.1

[redacted]

This document contains the high level design of the eStream Application Builder. The Builder is used to "prepare" an application before it can be eStreamed. This document describes the high level design of the application installation monitoring, file relocation and mapping, gathering of the initial profiling information of an application, the packaging of the eStream Set, and the merging of the newly uploaded user profile data.

Note: all references to "user" should be understood to mean the user of the Builder (i.e. the person who is responsible for creating eStream sets) and not the end-user of eStream technology.

This document described these steps involved in the preparation of the application: Installation Monitoring, Application Profiling, and eStream Packaging.

Modules

Installation Monitor:

- When the application is installed, we need to monitor the installation to see various "things" taking place on the computer. These could be:
  - Various updates to the System Registry
  - Files added to the Install directories (i.e. directories where application bits are copied as specified by the installing user). Lets call this group F_I.
  - Files added/updated to the Shared directories (e.g. "Program Files\Common Files"). Lets call this group F_C.
  - Files added/updated to the System directories (e.g. "WinNT\System32"). Lets call this group F_S.
  - Files added/updated to the User specific directories (e.g. "Documents and Settings\spujare\Application Data"). Lets call this group F_U.

  Note that once this information is gathered by the "Installation Monitor", a single "Installation Set" is prepared where all the files are stored in a single directory hierarchy. Note that files in the F_C, F_S and F_U groups (i.e. F_{CSU} group) are also stored here. For these files a "mapped location" is created under the single directory hierarchy. The Installation Set typically creates a map of all files (called ISM for Installation Set Map) described above with each entry containing the following info:

  1. fileId for the file
  2. location and name of the file. Note that the location will be the actual location for F_I files, but mapped location for the F_{CSU} files.

- After we gather the above information, we need to prepare a "File Relocation Map" (FRM) that is used by the client file spoofer to spoof references to any file in the common file group (i.e. F_{CSU}). For example: when the eStreamed app makes a reference to a file `C:\Program Files\Word\Foobar`, the file spoofer actually redirects that reference to `Z:\Program Files\Word\Foobar`. It does that because of the File Relocation Map. Each entry in the FRM typically has the following info:
  1. fileId (which references an entry in the ISM).
  2. Actual location where the application expects it (i.e. `C:\Program Files\Word\Foobar)`.

Profile Module:
During the application building process, the Builder program queries the user for the name of the application executable. Then Builder program starts and terminates the application executable immediately to gather initial sequence of the application page access pattern. After the initial seed of profile data is acquired, the Profile Sequence Matrix is combined with other appInstallBlock data gathered from the Install Monitor.

Profile Sequence Matrix is a 2D matrix of a profile data. Each entry of the matrix [column C, row R] is an integer value indicating the number of times a page R is requested following the request of page C. This successor request pattern is the page requests missed in the eStream cache manager.

Package Module:
In the final phase of the Builder program, the appInstallBlock is encapsulated into a special installation executable and the application files is archived into a single compressed package. The install executable containing the appInstallBlock and the archive of application files can then be placed in a suitable eStream Set server for ASP to download to their machines.

Merge Module: (not supported in version 1.0)
During normal eStream application usage, the eStream client gathers profile information for that particular run of the application. Then at the termination of an eStream application, it uploads the new Profile Sequence Matrix to the Profile Server. The clients should not upload the Profile Sequence Matrix from previous runs because the Profile Server has no mechanism for distinguishing between previously uploaded data and the newly acquired data.

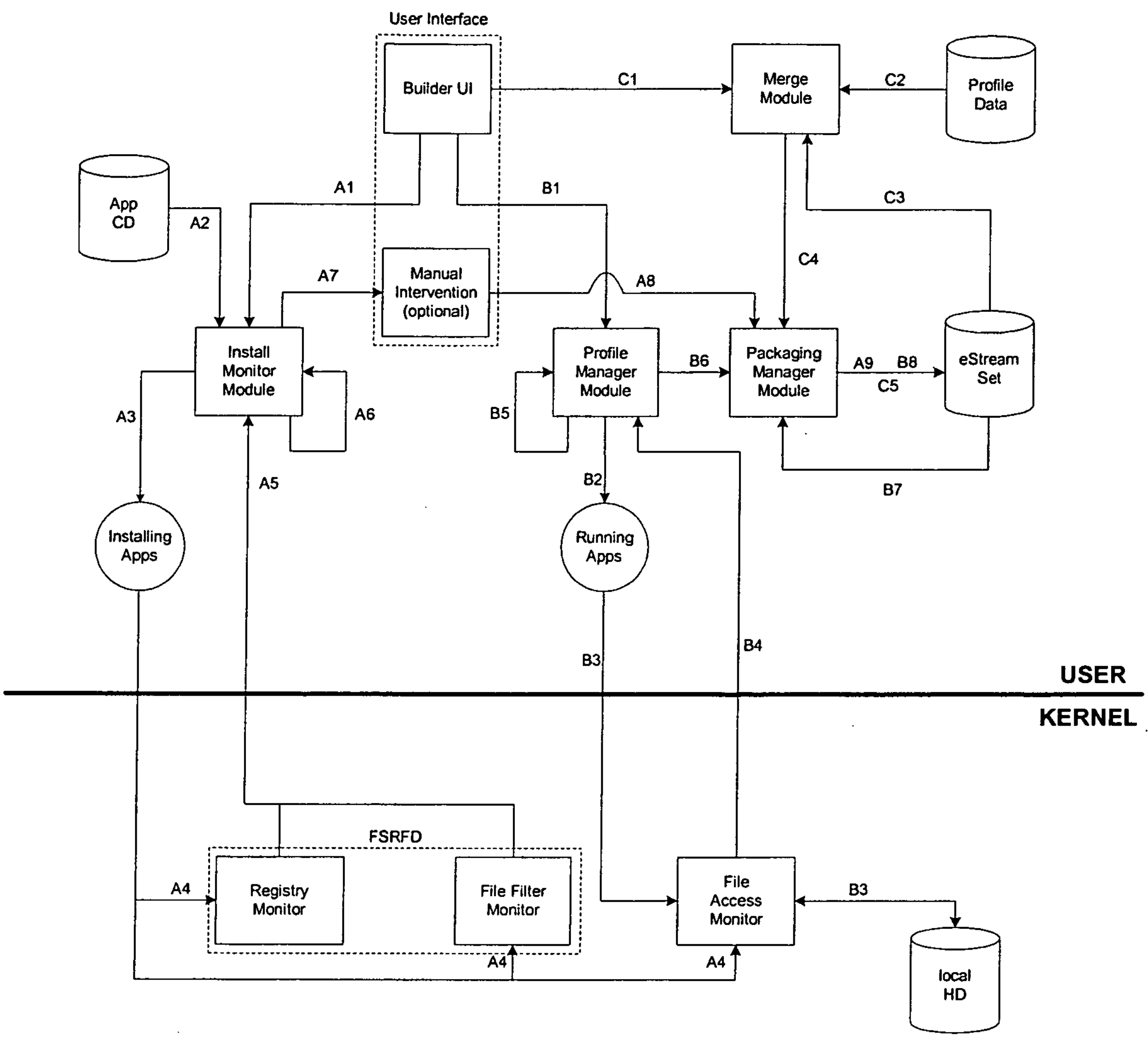
At appropriate time, the Builder is invoked to merge the newly uploaded per-user Profile Sequence Matrix into a collective Matrix. The merging algorithm may be designed with some heuristics to prevent the data biasing toward power users. This collective Matrix can be reinserted into the appropriate appInstallBlock then downloaded by any requesting eStream clients.

Kernel Device Drivers:

In addition, kernel device drivers are used to actually hook into the operating system to monitor the registry and file changes during installation of the application. This is accomplished by the FSRFD module.

The kernel device driver is also used for gathering monitoring file block references from the operating system to the file system. This is accomplished by the File Access Monitor.

eStream Application Builder High-Level Design Diagram



v 0.1

Interfaces

The interfaces are divided into three use cases: application installation monitoring, application profiling, and merging of the uploaded profile data.

Use Case #1: Install Monitor

A1. Builder UI to Install Monitor – send the name of the application installation executable

A2. App CD to Install Monitor – the CD containing the application is fed into the installation monitor module

A3. Install Monitor to Installation App – invoke the installation program

A4. Installation App to FSRFD – monitor all changes to the registry and files when installation program write to local file system

A5. FSRFD to Install Monitor – send all registry and file changes

A6. Install Monitor to itself – repeat all applications in the suite and merge all data

A7. Install Monitor to Manual Intervention – send a list of registry and file captured by the install monitor to UI and allow user to add or delete any entries

A8. Manual Intervention to Package Manager – send the final registry and file relocation data to the packager

A9. Package Manager to database – data set is packaged into appInstallBlock and the rest of the application files suitable for eStreaming

Use Case #2: Profiling

B1. Builder UI to Profile Manager – send the name of the application executable

B2. Profile Manager to Run App – invoke the application

B3. Run App to eStream File Access Monitor – record sequences of page requests

B4. File Access Monitor to Profile Manager – save the profile information

B5. Profile Manager to Profile Manager – repeat for each application in the suite

B6. Profile Manager to Package Manager – send all profile data for merging into a single data

B7. database to Package Manager – get eStream Set from the database

B8. Package Manager to database – save the updated eStream Set

Use Case #3: Merging Profile data (not supported in version 1.0)

C1. Builder UI to Merger – send the application name with profile data to merge

C2. database to Merger – get uploaded Profile Sequence Matrix from the Profile Server

C3. database to Merger – get the old appInstallBlock from database

C4. Merger to Package Manager – reinsert the Profile data into appInstallBlock

C5. Package Manager to database – save the updated appInstallBlock

Requirements

Please see eStream1.0-REQ.doc for the most up-to-date list of the Builder requirements. This requirement list may not contain the most recent changes. Each requirement is identified by a tag such as R-XXXX for easy references elsewhere in the document.

- **R-Background**: The installation monitor runs in the background, when an eStream application is installed as part of its preparation or building.
- **R-RegistryCapture**: The installation monitor captures all the updates to the System Registry that take place during the install. These updates are captured as a .REG file. Note that registry key deletions are also captured and stored in the .REG file. Please see the LLD doc by Charles Booher about the Registry spoofing database.
- **R-FileCapture:** The installation monitor records all the files created in the two kinds of directories: the install directory (the F_I group described above) and the common directories (the F_{CSU} group). All the files created are copied to the Installation Set and the File Relocation Map (FRM) created for the F_{CSU} group files. <Note: as far as a system common DLL is concerned, the eStream client should (a) overwrite the existing DLL if it exists (b) spoof it if doesn't exist. This is necessary because some installations may depend on newer versions of, say MSVCRT.DLL and in Windows there is no way to maintain different versions of the same DLL>.
- **R-InitialProfiling:** The Builder must be able to gather initial set of application profile data. This data consists of the page access pattern for starting and immediately shutting down an application.
- **R-Packaging:** The Builder must package the eStream Set into a easily manageable packages suitable for ASP administrators to download to their servers. The package can be divided into two sets:

  1. Installation Set - an appInstallBlock which is a set of data needed to setup the client machine for running a particular eStream application. The appInstallBlock is converted into an installation executable for simplifying the initial application set-up on the client machine.
  2. Run-time Set - a set of files associated with a particular application. At run-time, appropriate pages from this set of files is streamed to the client.

- **R-Merging (not supported in version 1.0):** The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock. This profile data can also be collected for use by the ASP or application developers.
- **R-NoQuietOperation:** The Builder is not required to be run in an environment where no other applications are running. But, since the Builder operates by invoking application installation program, it inherits any restrictive "Quiet-Operation" requirement from the installation program. Thus, if the installation program of an application has a "Quiet-Operation" requirement, then the "Quiet-Operation" must be enforced by the user when running the Builder.
- **R-AllClient**: The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support. <Preferably there should be only one builder program that should recognize the OS it is running on and should create the appropriate installation set. Also if possible, we should be able to "diff" installation sets for different OSs and if they are same, we should be able to create

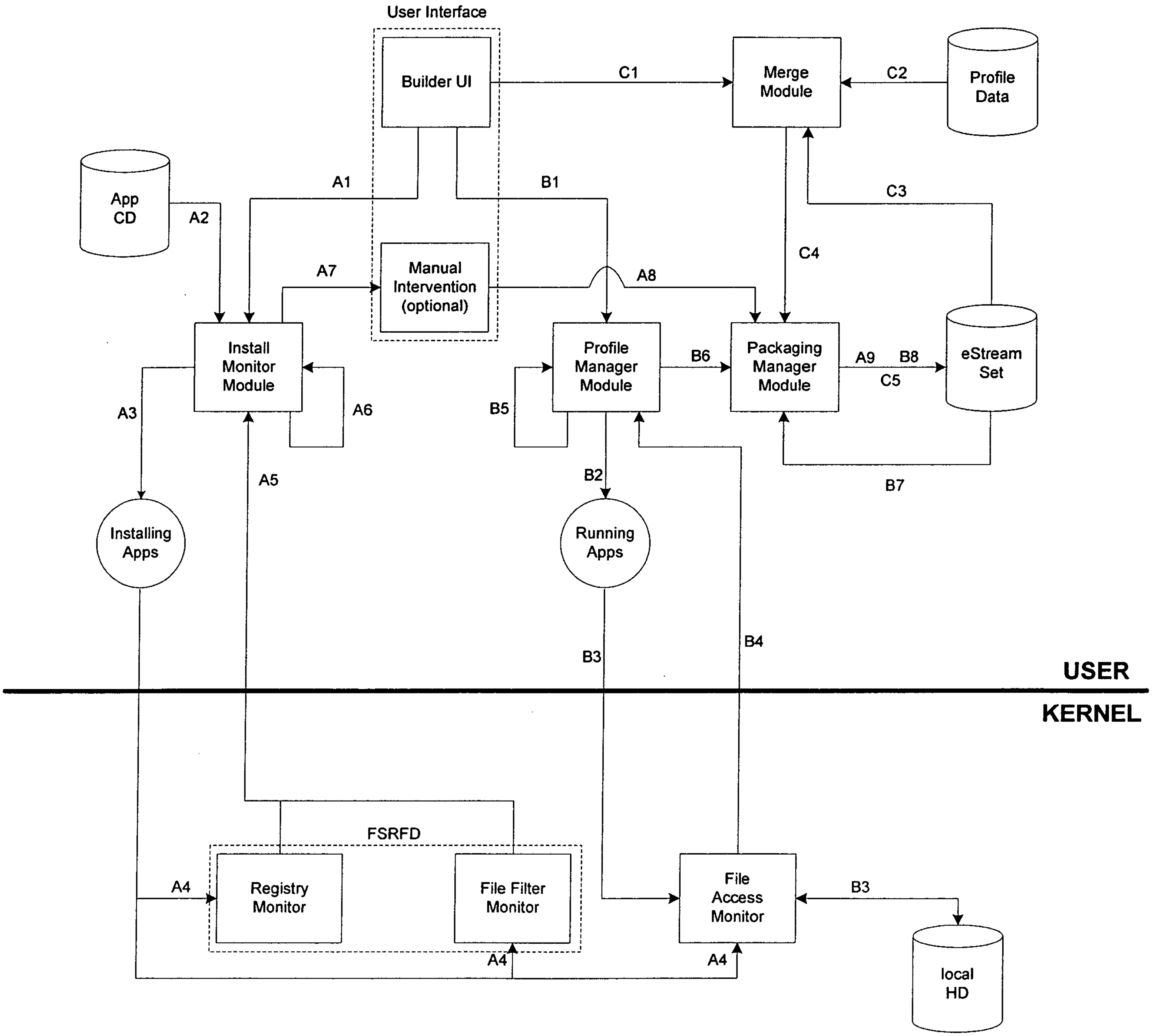
a single installation set for those OSs. The clients to be supported are W2K, WinNT4.0 and Win98>.

- **R-AppIdGeneration**: It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it. Typically the builder will generate a default appId number for a new application which can be overridden by the ASP installer by using a Builder tool.
- **R-SuiteSupport**: It should be possible to create a merged eStream set for a suite of applications. E.g. Office consisting of Word, Excel and Powerpoint. This could be done either by providing a tool for merging multiple eStream sets or by allowing the builder to serially monitor multiple installations in a session and then allowing the user to create a single package at the end of the session.
- **R-Testing**: It should be possible to test the Builder using a stand-alone tester and not require the eStream client+server programs.
- **R-UpgradeSupport**: The appInstallBlock should have support for indicating upgrades at the support site. E.g. When an eStream application is upgraded at the server (not as a separate app), the client will no longer be able to access/use it. We should provide some version of the appInstallBlock itself so that clients should detect that they will need to download the appInstallBlock again.
- **R-ManualIntervention**: In the process of creating an eStream set it should be possible for the user to delete file entries and registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.

Issues

- Profile Sequence Matrix is different for different machine configuration even if the user's usage pattern is the same.
- Profile Sequence Matrix doesn't contain the right successor profile information as eStream cache is warmed up and pages from the cache is replaced.
- Merging Module must take different machine configuration into account. Should this information be uploaded by the client as the same time it uploads the Profile Sequence Matrix to the Profile Server?
- What is the difference between profiling based on the page sequencing seen by the eStream Cache Manager versus the page sequencing missed by the eStream Cache Manager?

eStream Application Builder High-Level Design Diagram



v 0.1

Tricky Builder Issues

Author: Sanjay Pujare

This document enumerates all those tricky issues that may make the Builder's job difficult. Even though some solution may be proposed for some issues, not every issue would have a solution described in this document. The purpose of this document is mainly to keep track of Builder issues that may impose some limitations on the eStream technology. This way Omnishift marketing and deployment are aware of these limitations.

1) The Builder cannot capture updates to existing files in an intelligent fashion (i.e. if the updates are based on a context or existing contents, it is very difficult to capture that). So the current Builder will just flag an error, if such an update occurs.

   Solution
   - These updates are probably very rare, so we can defer it to the next release.
   - For this release, we can try to solve this on a case by case basis e.g. we will try to solve this issue for INI files.
   - Based on our understanding of general app installations, we might be able to make some generalizations that we can use in eStream e.g. Only certain files get updated; there is a definite pattern of updates.
2) The current Builder drivers are based on the NT driver model, and hopefully we can implement the same functionality in the Win98 drivers, but this needs to be ensured. (This shouldn't be an issue, but…)
3) We need to think some more about those cases when device drivers are installed by apps. Issues that can arise:
   - This may not work correctly on the eStream client, just because the driver installation didn't take place properly.
   - The Builder would need to be able to figure out in an automated way if a client reboot is required or not.
   - If the driver installation is h/w or s/w specific that can be difficult to tackle.

   Solution
   - As we eStream more apps and gain more experience, we should be able to figure out solutions.
4) There could be an ambiguity when the Installmon is trying to change absolute paths (or absolute values in general) to relative paths. e.g. A path like C:\WINNT can be changed to %SystemRoot% or %windir% since both of those environment variables are set to "C:\WINNT" on my system.

   Solution
   - We can prioritize env vars and registry keys as described in the BuilderUI-LLD design document.

- We should encourage Builder operators to use as distinct values as possible for env variables and registry keys for the Builder machine.

eStream Set Format Low Level Design

Sanjay Pujare and David Lin
Version 0.7
[redacted]

Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

Note: Fields greater than a single byte is stored in little-endian format. The eStream Set file size is limited to 2^64 bytes. The files in the CAF section are laid out in the same order as its corresponding entries in the SOFT table.

Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.
- **Flags [4 bytes]:** Flags pertaining to EStreamSet
- **Reserved [32 bytes]:** Reserved spaces for future.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- **VendorNameLength [4 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.

- o **AppBaseNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- o **AppBaseNameLength [4 bytes]:** Byte length of the application base name.
- o **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.
- o **MessageIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- o **MessageLength [4 bytes]:** Byte length of the message text.
- o **Message [X bytes]:** Message text. Null-terminated.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- o **NumberEntries [4 bytes]:** Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

Root Version structure: (variable number of entries)

- o **VersionNumber [4 bytes]:** Version number of the root directory.
- o **FileNumber [4 bytes]:** File number of the root directory.
- o **VersionNameIsAnsi [1 byte]:** 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.
- o **VersionNameLength [4 bytes]:** Byte length of the version name
- o **VersionName [X bytes]:** Application version name. I.e. "SP 1".
- o **Metadata [32 bytes]:** See eStream FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1. The start of the SOFT table is aligned to 8 bytes boundary for faster access.

SOFT entry structure: (variable number of entries)

- o **Offset [8 bytes]:** Byte offset into CAF of the start of this file.

- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

b. *AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. *EStream Directory*

An eStream Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream directory file.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.

- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.
- **NumEntries [4 bytes]:** Number of entries in the directory. Some entries are used for storing long file names and some are unused due to deleted files. So the NumEntries must be equal or less than NumFiles.

Fixed length entry for each file in the directory consists of 2 formats (short format for storing files with name that fit the 8.3 convention; and long format for storing long file names). Each entry is 84 bytes and the entry are aligned on every 4K page boundry. Thus, in the first 4K page of the directory, the padding consists of 12 unused bytes (52 bytes for header + 48 entries * 84 bytes per entry + 12 unused bytes = 4096 bytes). In all subsequent pages, the padding is 64 bytes (48 entries * 84 bytes per entry + 64 unused bytes = 4096 bytes):

Short Filename entry:

- **Format [1 byte]:** Format of this entry, should be 's' for short format, 'l' for long filename format, or possibly 'u', for unused.
- **ShortLen [1 byte]:** Length of short file name.
- **LongLen [1 byte]:** Length of long file name.
- **UNUSED [1 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the short file name. Algorithm TBD.
- **ShortName [24 bytes]:** 8.3 short file name in unicode
- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **eStream flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user
  - **Bit 2:** Directory – Set if the file is an eStream Directory
  - **Bit 3:** Archive – Set if the file is an archive
  - **Bit 4:** Normal – Set if the file is normal
  - **Bit 5:** System – Set if the file is a system file
  - **Bit 6:** Temporary – Set if the file is temporary

  The bits of the **eStream flags** have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
  - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
  - **Bit 2:** Read-only – Set if the file is read-only

Long Filename entry:

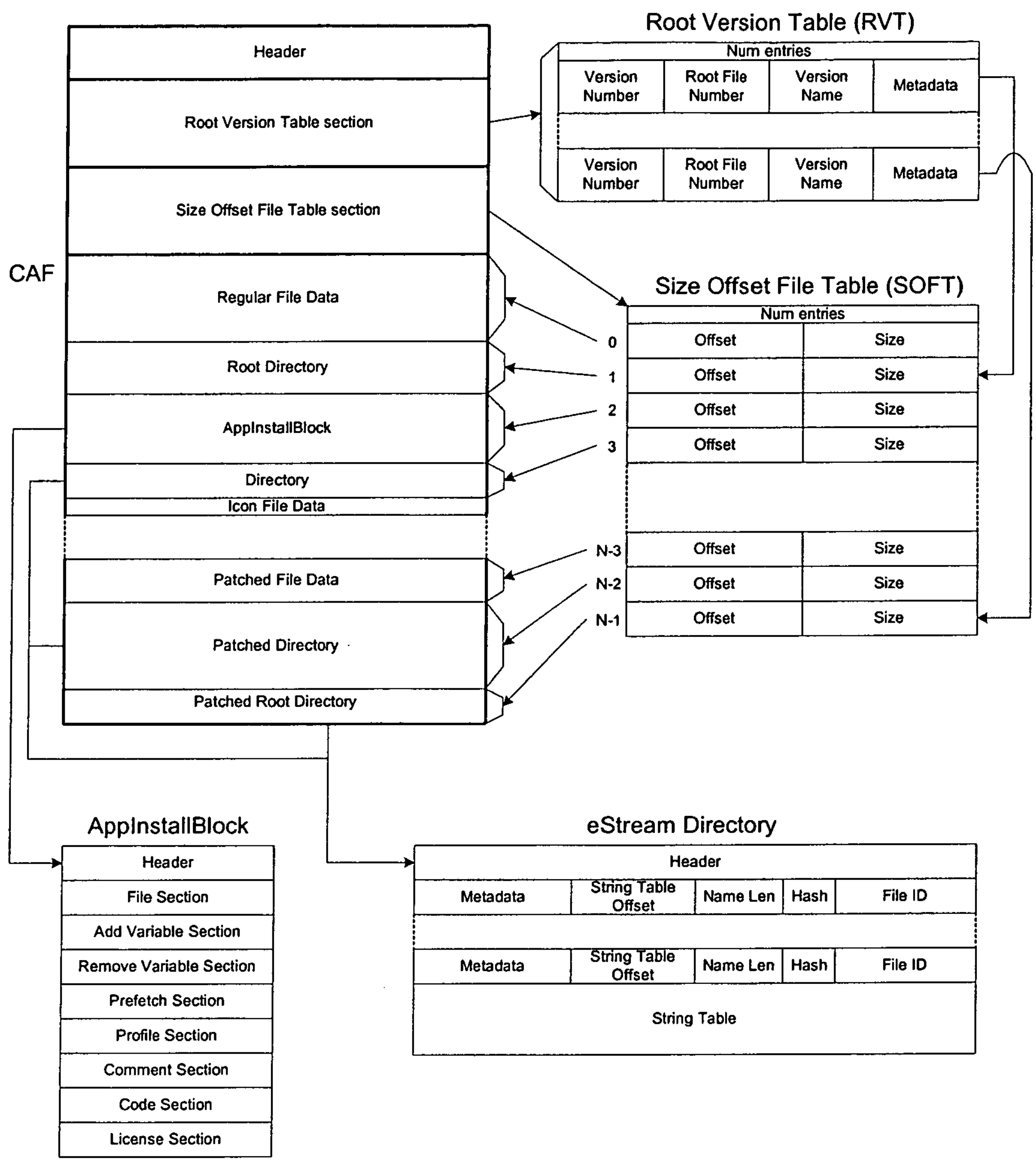
- **Format [1 byte]:** Format of this entry, should be 'l' for long filename format.

- **Index [1 byte]:** Number of this entry out of those used for this file's long name.
- **UNUSED [2 byte]:** Padding
- **NameHash [4 bytes]:** Hash value of the long file name. Algorithm TBD.
- **LongName [76 bytes]:** Long filename in Unicode format.

d. Icon files

- **IconFileData [X bytes]:** Content of an icon file.

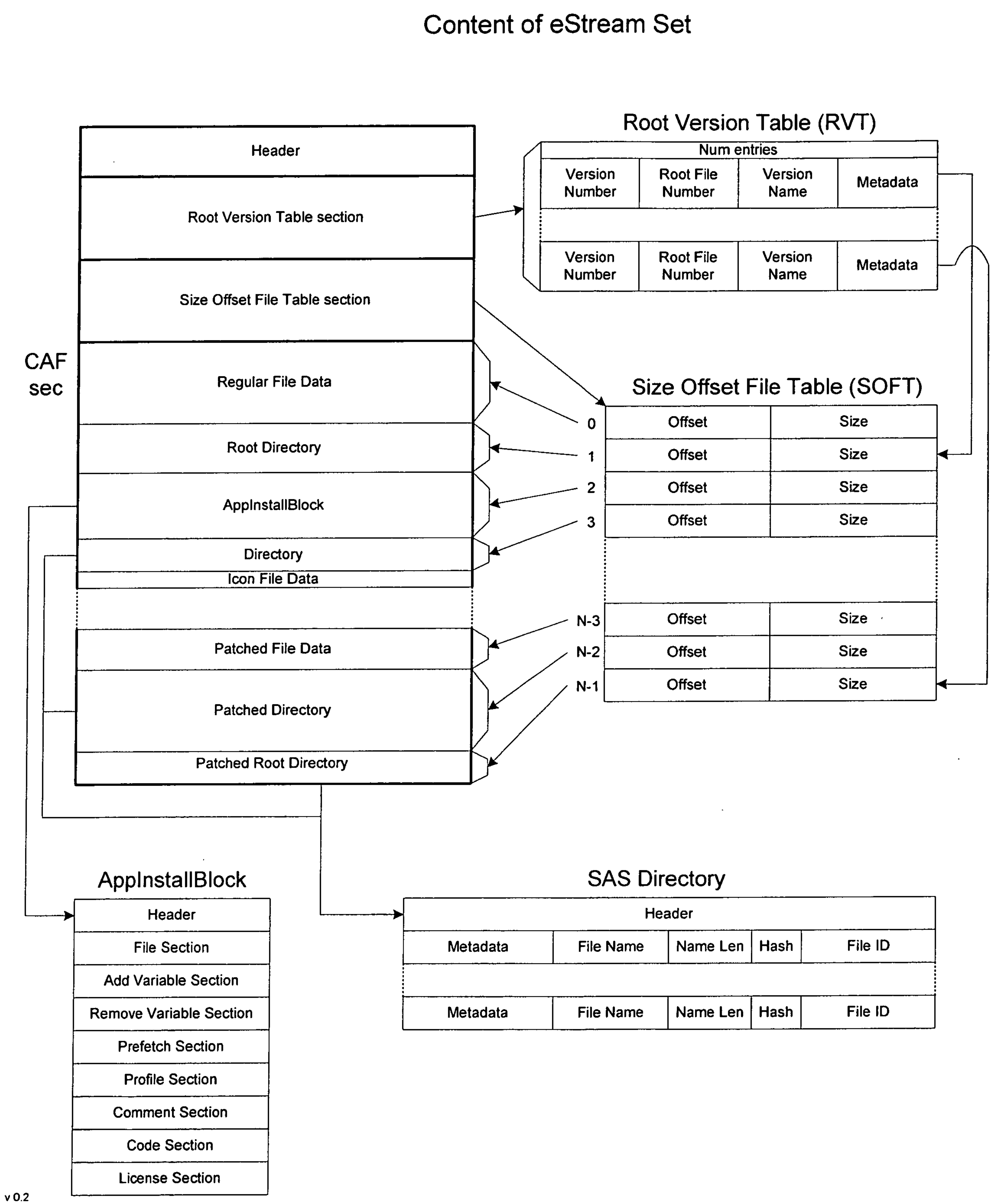
Format of the eStream Set

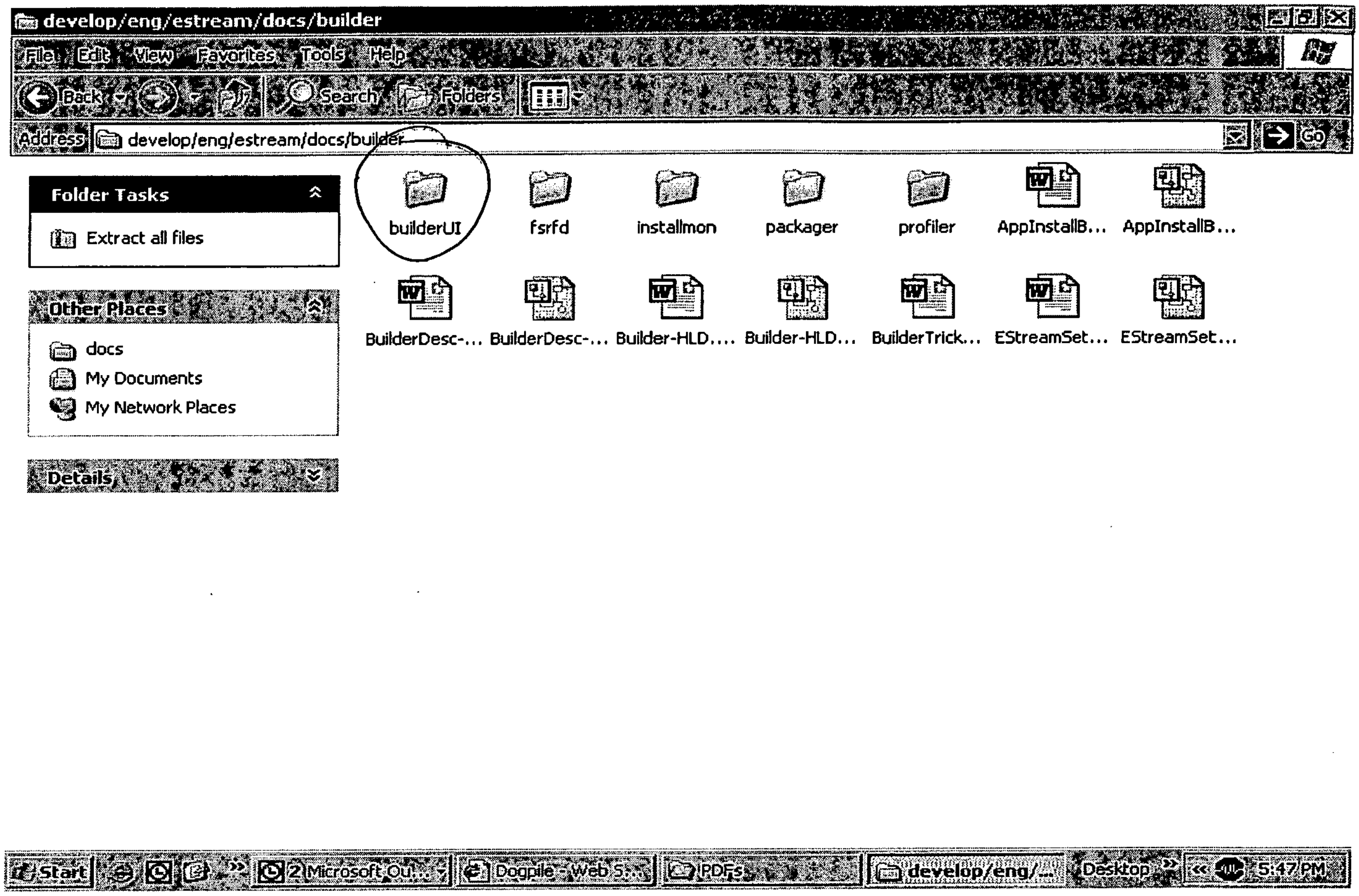


v 0.2

Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.


Content of eStream Set
Root Version Table (RVT)
Num entries
Version Number
Root File Number
Version Name
Metadata
Version Number
Root File Number
Version Name
Metadata
CAF sec
Header
Root Version Table section
Size Offset File Table section
Regular File Data
Root Directory
AppInstallBlock
Directory
Icon File Data
Patched File Data
Patched Directory
Patched Root Directory
Size Offset File Table (SOFT)
0
1
2
3
N-3
N-2
N-1
Offset
Size
AppInstallBlock
Header
File Section
Add Variable Section
Remove Variable Section
Prefetch Section
Profile Section
Comment Section
Code Section
License Section
SAS Directory
Header
Metadata
File Name
Name Len
Hash
File ID
v 0.2



[/STANDARD.01] [redacted]

eStream BuilderUI Low Level Design

Sanjay M Pujare
<Date>

Functionality

The BuilderUI is the user interface part of the Builder. The operator uses this interface to use various functions provided by the Builder. Note that this UI may or may not be a graphical user interface. This low-level design is based on the assumption that a graphical user interface is not necessary.

Data type definitions

Interface definitions

Component Design

When the Builder UI is invoked with command line arguments which indicate that this was invoked by the `Runonce` mechanism of Windows, the control is transferred to the `InstallMon::startCaptureAfterReboot()` function with the command line arguments passed as arguments to the function. When the Builder is invoked normally, it presents a menu which is managed by the function `MainMenu`.

MainMenu

```
This function manages the following menu hierarchy. Each
menu option (leaf node) is followed by a function name in
parentheses that is called to handle the option.
1) eStream Set Menu
   1) New eStream Set (NewEStreamSet)
   2) Open eStream Set (OpenEStreamSet)
   3) Save New/Upgraded eStream Set (EstreamSetCreation)
2) Monitoring Menu
   1) Start Monitor (StartMonitor)
   2) Stop Monitor (StopMonitor)
   3) Check Status (CheckStatus)
   4) Inform Machine Reboot (InformMachineReboot)
   5) Get and Resolve Registry Set (GetRegistrySet)
   6) Get and Resolve Files Set (GetFilesSet)
3) Profiling Menu
   1) Set the location of app executable (GetAppPath)
   2) Gather Initial Profile (GatherInitialProfile)
4) eStream Set Creation Menu
   1) Set custom DLL (GetCustomDLL)
   2) Set User Comment (GetUserComment)
   3) Set environment variables (GetEnvVars)
```

```
4) Set Reboot flag (GetRebootFlag).
5) Set License Agreement (GetLicenseAgreement).
```

NewEStreamSet

```
{
  If there is an existing eStream set that hasn't been
  saved, warn the user.
  Get the following values from the user:
    • Name of the app setup program in gAppSetup
    • Dest directory where app will be installed in gDest-
      Dir (provide a default value).
    • Dest location to store the new eStream set in
      gDestEstreamPath (provide a default value).
}
```

OpenEStreamSet

```
{
  If there is an existing eStream set that hasn't been
  saved, warn the user.
  Get the following values from the user:
  • Location of the existing eStream set in gSrcEstream-
    Path (provide a default value).
  • Whether the user wants to create an upgrade from this,
    or just wants to change the existing eStream set (gUp-
    grade)
  • If this is an upgrade (gUpgrade is true), get all the
    values obtained by NewEstreamSet (i.e. gAppSetup,
    gDestDir, gDestEstreamPath).

  Read the eStream set pointed to by gSrcEstreamPath;
  Load the existing file tables in gSrcCopiedFiles,
  gSrcSpoofedFiles and gSrcEFSFiles arrays;
}
```

EstreamSetCreation

```
{
  If there is no working eStream set, give error and re-
  turn.
  if (gUpgrade) {
    Call UpgradeEStreamSet() with appropriate arguments;
  }
  else {
    Call CreateEStreamSet() with appropriate arguments;
  }
  if (gDestEstreamPath is not set) {
    assert(this is an update of an existing eStream set
```

```
        and not an upgrade or a new eStream set creation);
      gDestEstreamPath = gSrcEstreamPath;
    }
    Save the eStream Set from memory to file to
      gDestEstreamPath;
}
```

StartMonitor

```
{
    If there is no working eStream set, give error and re-
    turn.
    if (gUpgrade) {
      Combine the gSrcSpoofedFiles and gSrcEFSFiles into an
      array fileTableArray as expected by startCapture
      below;
    }
    Call InstallMon::startCapture(gAppSetup, gDestDir,
        gUpgrade, fileTableArray);
}
```

StopMonitor

```
{
    If monitoring wasn't started, give error and return.
    Call InstallMon::stopCapture();
}
```

CheckStatus

```
{
    Ensure that monitoring was started and not stopped
    Call InstallMon::checkSetupStatus();
}
```

InformMachineReboot

```
{
    Ensure that we are in the middle of monitoring.
    Call InstallMon::machineToBeRebooted();
}
```

GetRegistrySet

```
{
    Call InstallMon::getRegistryList();
    Store the set in gNewRegistry data structure;
}
```

GetFilesSet

```
{
    Call InstallMon::getFilesList();
```

```
    Store the set in gNewFiles data structure;
    Also we need to capture changes made to INI files; since
    this has not been taken care of in the app install block
    and other parts of the Builder+Client we will need to
    make changes in all those components which are affected.
}
```

GetAppPath

```
{
    Ensure that all the InstallMon related data was captured.
    Get the location of the app that needs to be run
    to gather profiling info;
    Store it in gProfileAppPath;
}
```

GatherInitialProfile

```
{
    // this is the function that is used to get the initial
    // profile data (i.e. set of pages prefetched when an
    // eStream app is started for the first time)
    // implementation of this is yet to be defined
}
```

GetCustomDLL

```
{
    Get the location of the custom DLL file, validate that it
    is a DLL and store the path in gCustomDLLPath;
}
```

GetUserComment

```
{
    Get the user comment (optionally by browsing a text file)
    and store it in gUserComment;
}
```

GetEnvVars

```
{
    Call the InstallMon::setEnvVars() function;
}
```

GetRebootFlag

```
{
    Until we come up with an algorithm to determine if a re-
    boot is required for an eStream app, get this value from
    the user. The default is FALSE: we do not want to reboot
    the client PC when the user subscribes to this eStream
    app.
```

```
}
```

GetLicenseAgreement

```
{
  Get license agreement from the user. This could be:
    □ either, a default OmniShift license agreement
    □ or, a default ASP agreement
    □ or, license agreement that the app displayed.
  Let the user decide and enter the proper one. Provide a
  default based on our policy.
}
```

Interesting issues to deal with:

Testing design

Unit testing plans

Testing of the UI itself is a comparatively trivial task. The testing will basically consist of traversing the whole menu hierarchy. Since the menu is similar to a typical File Open -> File Edit -> File Save kind of a user application, this can be tested using simple hooks.

Stress testing plans

Since the Builder will be used in house at least initially only simple stress testing should be necessary. Make sure that Builder doesn't crash in the middle of processing so that we don't lose important data. Performance is not considered to be important.

Coverage testing plans

Basically the following 3 paths will be exercised:

1. Create a new eStream set
2. Open an existing eStream set to modify some data in it
3. Open an existing eStream set to create an upgrade for it

Cross-component testing plans

Will be tested as a component of the whole builder.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

- We need to think about the problem of converting absolute file paths discovered in the monitoring process to paths relative to some application or system registry key. Although most cases may not present a problem, we may have some difficult cases, which may make this problem non-automatable i.e. we may need some user intervention. Consider the case:

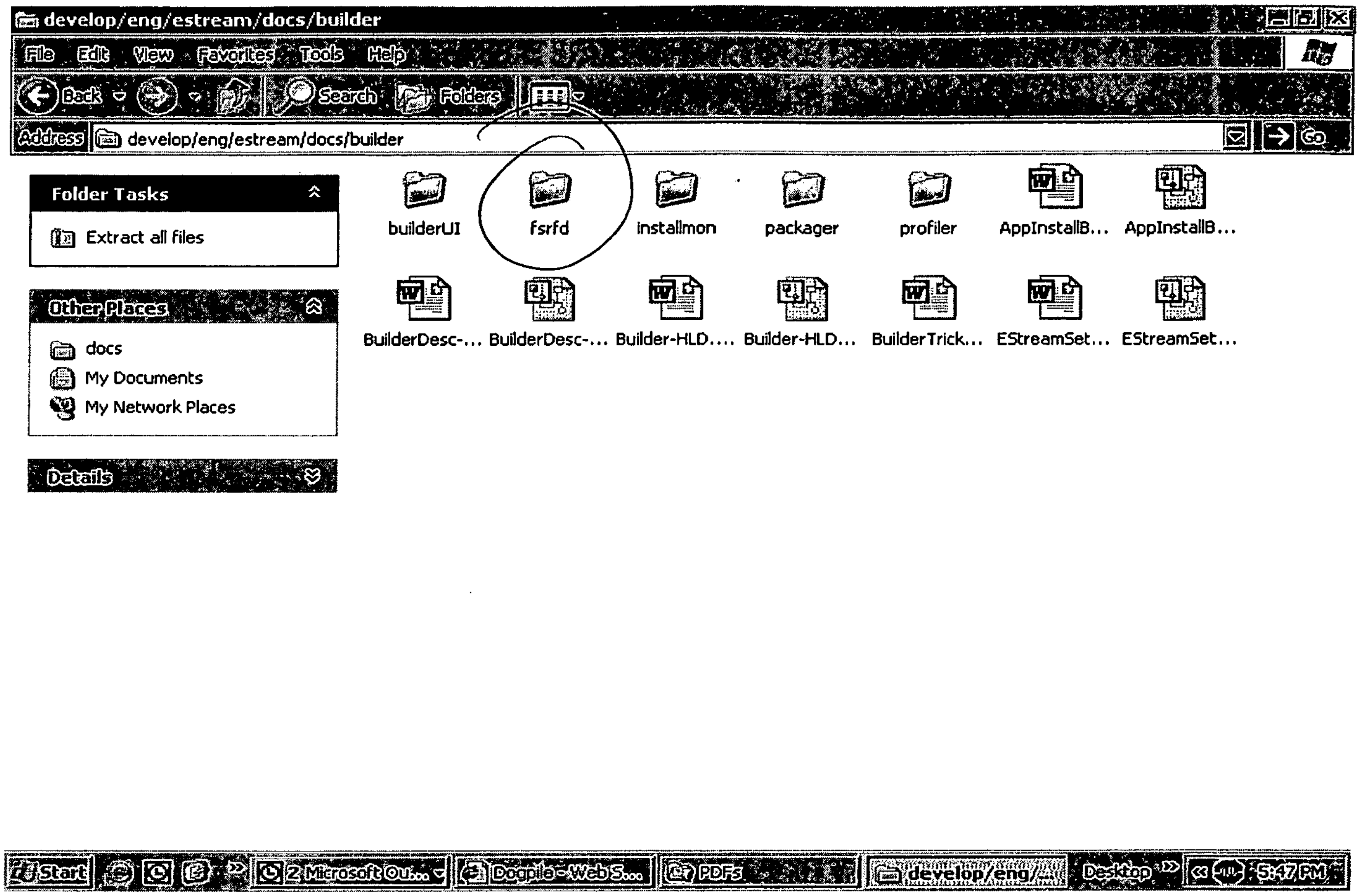
  KEYONE = C:\FOO\BAR

  KEYTWO = C:\FOO

  KEYTHREE = BAR

  If we notice that a file was copied to C:\FOO\BAR it won't be possible to convert this absolute path to a unique relative path since there are 2 solutions possible: %KEYONE% or %KEYTWO%\%KEYTHREE%.

  The way to solve this is by tracking only a set of well-known environment variables and registry keys. Also in this set we prioritize all of them. So in the above case, %KEYONE% will be preferred over %KEYONE%\%KEYTHREE% just because of the way they were prioritized.



[/STANDARD.01]

[redacted]

eStream FSRFD Low Level Design

Sanjay M Pujare
[redacted]

Functionality

The File System and Registry Filter Driver (FSRFD) is a part of the Builder module that monitors file system and registry updates initiated by the Builder process. This driver just intercepts such requests and records them and returns the recorded data to the Install Monitor (*INSTALLMON* described in another LLD) program when requested by the latter. All the intelligence, such as any decision-making logic, resides in the INSTALLMON.

For registry updates such as add or modify, the FSRFD needs to record the value added or modified. For registry deletes only the value name needs to be recorded. For file updates, there is no need to record the file contents added or modified, since the Builder would be interested only in the final contents of a file.

Note:

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**
2. **The FSRFD will be used to monitor only one install at a time to simplify the design of the FSRFD. That means you cannot invoke multiple instances of the Builder at a time to monitor multiple installations. All Builder invocations on a machine have to be strictly sequential.**
3. **This design is based on the driver model for WinNT and Win2K. The Win98 driver is not covered here (yet).**

Data type definitions

The following struct is used to communicate information related to activating the FSRFD. Specifically, the process-id of the INSTALLMON and the 2 drives whose accesses need to be monitored are passed.

```
struct MonitorActivate_t {
  ULONG processId; // PID of INSTALLMON
  UCHAR sysDrive;  // System drive letter
  UCHAR destDrive; // Dest drive letter
};
```

The following struct is used to return monitored data back to the INSTALLMON. Note that this is a variable size struct where the last field keyName is an array of one wide-char, but in reality is an array of length whose value is the sum of 3 length fields in the struct (nameLength, valueNameLength and dataLength).

```
struct IMON_ENTRY {
    UCHAR regOrFile; // 'R' for registry, 'F' for files
                     // and 'E' for end of data
    UCHAR updateType; // 'A' for add, 'D' for delete,
                      // 'U' for update
    UCHAR valueType;  // for registry only: value type
// REG_SZ, REG_DWORD, REG_BINARY,
// REG_DWORD_LITTLE_ENDIAN, REG_DWORD_BIG_ENDIAN,
// REG_EXPAND_SZ, REG_LINK, REG_MULTI_SZ, REG_NONE,
// REG_QWORD, REG_QWORD_LITTLE_ENDIAN,
// REG_RESOURCE_LIST
    ULONG nameLength; // length of name (file or
                      // registry key) in wchars
    ULONG valueNameLength; // length of value name (if
                           // it exists) in wchars
    ULONG dataLength;     // length of data in bytes
    WCHAR keyName[1];    // keyName followed by
                        // valueName followed by
                        // data: note none of these are
                        // null terminated & are wide
                        // chars
};
```

Note about the updateType field: 'A' is used for file creation and 'U' is used for any updates to the file. So if a 'U' is seen without an 'A' for a file that means the file was modified but not created in this session.

The following struct is used as the device extension in the FSRFD devices. Note that this extension is used for all device objects: the device that is created in the DriverEntry function to represent an "INSTALLMON" device for the INSTALLMON to access our driver as well as the devices that are created to create a filter layer above existing drives.

```
enum DEVICE_TYPE {
        INSTALLMONINTERFACE,
        STANDARD
};
```

```
struct DeviceExtension_t {
        PDEVICE_OBJECT deviceObject; // device
                       // for lower layer
        DEVICE_TYPE type; // see design
        KMUTEX pDeviceMutex;
        ProcessIdList
        // pointer to list of process-ids we are
        // interested in monitoring
        EntryList
        // This is the list that stores all the
        // info captured by the FSRFD until each
        // entry is queried by INSTALLMON
};
```

There is an array for FastIo that stores all the FastIo function pointers which is required in a file system filter driver such as this. Note that we need to provide entry points for all (or most?) of the FastIo routines in the dispatch table, since we need to pass the request down to the lower layer driver even if we are not interested in intercepting the request. For only some of the requests (e.g. all the FASTIO_*WRITE* requests), we would be recording the file access and creating an entry to be returned to INSTALLMON. The Component Design section describes in more detail the FastIO routines that are implemented by this driver.

Interface definitions

INSTALLMON interfaces

Since the FSRFD is a driver, it cannot provide directly callable APIs. Instead the INSTALLMON communicates with the FSRFD using the DeviceIoControl Win32 API. It uses Ioctl codes MON_ACTIVATE, MON_DEACTIVATE and MON_GET_ENTRY. The DeviceIoControl API looks as follows:

```
BOOL DeviceIoControl(
  HANDLE hDevice,              // handle to device
  DWORD dwIoControlCode,       // operation control code
  LPVOID lpInBuffer,           // input data buffer
  DWORD nInBufferSize,         // size of input data buffer
  LPVOID lpOutBuffer,          // output data buffer
  DWORD nOutBufferSize,        // size of output data buffer
  LPDWORD lpBytesReturned,     // byte count
  LPOVERLAPPED lpOverlapped   // overlapped information
);
```

The semantics of each of the MON_* codes is defined below. Note that this is described from the caller's (i.e. INSTALLMON) point of view.

Ioctl 1 – MON_ACTIVATE

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_ACTIVATE

lpInBuffer:
Address of MonitorActivate_t struct. This contains the process id of INSTALLMON.

nInBufferSize:
Sizeof(MonitorActivate_t)

Output:

lpOutBuffer:
Should be a ptr to a ULONG (at least).

nOutBufferSize:
Size of above buffer

lpBytesReturned:
Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_ACTIVATE is sent to the FSRFD when the INSTALLMON wants to start monitoring an installation. MON_ACTIVATE can be sent only when the FSRFD is not already active - either after the driver is loaded the first time or after the last MON_DEACTIVATE request.

Errors:

- STATUS_ALREADY_ACTIVE: FSRFD was already activated. The processId of the old activation is returned in the ULONG pointed by lpOutBuffer.
- STATUS_INVALID_ARG: One of the arguments passed is not valid (either invalid MonitorActivate_t ptr or processId).

- STATUS_INVALID_DRIVE: Either the sysDrive or the destDrive (or both) is invalid.

Ioctl 2 – MON_DEACTIVATE

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_DEACTIVATE

lpInBuffer:
NULL, or pointer to a ULONG where the ULONG has a non-zero value indicating a "forced" deactivation. A "forced" deactivation is done when the FSRFD still has entries that are not going to be retrieved.

nInBufferSize:
0 or size of ULONG (depending on the above).

Output:

No need for OUT arguments.

Comments:

MON_DEACTIVATE is sent to the FSRFD when the INSTALLMON wants to stop monitoring an installation. MON_DEACTIVATE can be sent only when the FSRFD is already active i.e. after the last MON_ACTIVATE request.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was already deactivated. No special action is needed to be taken for this error condition. The caller can simply send a new MON_ACTIVATE.
- STATUS_PENDING_DATA: The FSRFD has some entries that were not read using

MON_GET_ENTRY. This error is only returned in case of non-forced deactivation.

Ioctl 3 – MON_GET_ENTRY

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_GET_ENTRY

lpInBuffer:
NULL.

nInBufferSize:
0

Output:

lpOutBuffer:
Should be a ptr to a IMON_ENTRY struct.

nOutBufferSize:
Size of above buffer

lpBytesReturned:
Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_GET_ENTRY is sent to get the next "entry" from the FSRFD. An "entry" is a record of a registry or file update intercepted by the FSRFD. The details are returned in the IMON_ENTRY struct passed in the lpOutBuffer argument. Note that the FSRFD uses an event object (created using the IoCreateNotificationEvent API) to signal the INSTALLMON that an "entry" is available to be read. INSTALLMON waits on this event object before retrieving the entry using MON_GET_ENTRY.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was not activated.
- STATUS_INVALID_ARG: One of the pointer arguments passed is not valid.
- STATUS_INSUFF_BUFFER: The buffer size indicated by nOutBufferSize is insufficient for the current "entry" data.

Ioctl4 – MON_GET_ERROR

We need this to indicate occurrence of an error whenever this occurs in any of the dispatch functions. We can either implement this control code or just return an error code for any MON_GET_ENTRY call that reflects that an error occurred.

Event Object interface

As mentioned above, an event object (lets call it IMON event object) will be used to signal the INSTALLMON that a new entry is available. This event object will be created in the FSRFD in the processing of MON_ACTIVATE ioctl, as:

```
ext->pEvent = IoCreateNotificationEvent(
  L"\\BaseNamedObjects\\INSTALLMONEVENT",
         ext->eventHandle);
```

Whenever the FSRFD has a new entry, it signals using the above event object as follows:

```
KeSetEvent(ext->pEvent, 0, FALSE);
```

Whenever INSTALLMON gets the next entry using MON_GET_ENTRY ioctl, the FSRFD resets the event (if the list of entries is going to be empty after this entry) as follows:

```
KeClearEvent(ext->pEvent);
```

Whenever a MON_DEACTIVATE is processed, the FSRFD will close the event as:

```
ZwClose(ext->eventHandle);
```

Kernel or Low Level Driver Interfaces

Every driver needs a `DriverEntry` routine that is called when the driver is first loaded. This routine for FSRFD is described in the Component Design section.

The FSRFD inserts "hook routines" that intercept relevant registry and file system calls. The Component Design section describes which hook routines are inserted and what they do.

Component Design

Global variables

pImonDevice

This global variable points to the device object created in the DriverEntry function for the "\\Device\\installmon" device.

DriverEntry

```
NTSTATUS DriverEntry(IN DriverObject, IN RegistryPath)
{
          IoCreateDevice for "\\Device\\installmon";
          pImonDevice = pDeviceObject returned above;
          ext = pImonDevice->DeviceExtension;
          ext->type = INSTALLMONINTERFACE;
          IoCreateSymbolicLink with
              "\\DosDevices\\installmon";
          for all IRP_MJ_* values upto
               IRP_MJ_MAXIMUM_FUNCTION {
            DriverObject->MajorFunction[IRP_MJ_*] =
                 ImonDispatch
          }
          Setup the unload driver function
          DriverObject->FastIoDispatch = address of
            our fast io dispatch table;
            Note that we are interested only in
            FAST_IO_*WRITE* routines for getting our
            entries, however we need to implement all
            of them to call lower layered drivers. All
            our FastIo funcs are called ImonFastIo*;
          Create the necessary mutexes;
          Use PsSetCreateProcessNotifyRoutine to set a
          process create callback routine
          ImonProcessCallback;
}
```

ImonProcessCallback

```
{
          // similar to ImonProcessCallback
          // This function is called every time a
```

```
                // process on the system is created or
                // deleted. We need to figure out (by
                // checking the parent id in our list)
                // if we need to add or remove this process
                // from our list
                lock the mutex for ProcessIdList
                if not activated just return;
                if (this is process create) {
                   Look up the parent process id in the
                   ProcessIdList
                   If present, add this process id to the
                      ProcessIdList
                }
                else {
                   if this process id is present then
                      remove the process id
                }
                release the mutex
        }
```

ImonDispatch

```
        {
                // similar to FilemonDispatch
                // Instead of registering a different
                // function for each IRP_MJ_* this function
                // is called for all of them and this one
                // dispatches the right on based on the
                // control code
                This gets called for all IRP_MJ_*;
                if the device extension type tells us
                   INSTALLMONINTERFACE
                      call ImonDeviceFunc
                else
                      call ImonHookFunc
        }
```

ImonDeviceFunc

```
        {
                // similar to FilemonDeviceRoutine
                // This function is called whenever an Ioctl
                // comes from the Installmon process that is
                // meant to be a command for this FSRFD.
                This is a request from the INSTALLMON using
                the INSTALLMON device. We would mainly be
                processing IRP_MJ_DEVICE_CONTROL, although
                ImonFastIoDeviceControl should have been
                called. So if we come here just call
```

```
            ImonFastIoDeviceControl
            Call IoCompleteRequest?
}
```

ImonHookFunc

```
{
            // similar to FilemonHookRoutine
            // This is the hook function that is called
            // for all the I/O requests that is made by
            // the I/O manager that need to go through
            // this driver (i.e. for those requests
            // that we are filtering).
            We are interested in recording:
               IRP_MJ_CREATE where the Irp->
                  Parameters.Create.Options indicates
                  a new file create (as opposed to an
                  existing file open)
               IRP_MJ_WRITE
            Note that we have to get the current
            process-id using PsGetCurrentProcessId and
            look it up in the ProcessIdList (note: you
            have to exclude the first process-id since
            that belongs to the Installmon and not the
            setup program) and only if that search is
            successful, record the entry
            Note that we need to get the filename from
               the FileObject using code similar to
               FilemonGetFullpath
            Also we should be recording the entry when
            the request is successfully completed. So do
            this in the completion routine in a manner
            similar to filemon.
            To record:
               create a relevant IMON_ENTRY record;
               Call ImonAddEntry with this record;
            Pass all Irps to lower layer driver using
               IoCallDriver and getting the lower layer
               device object from this device's
               ext->deviceObject
}
```

ImonFastIoDeviceControl

```
{
            // similar to both
            // FilemonFastIoDeviceControl and
            // ImonDispatchIoctl
            // This function gets called for all the
```

```
        // IOCTLs. If it is from Installmon, we need
        // to process the MON_* commands or else
        // just pass on the command to the lower
        // layer driver.
        Get the current device's extension
        if type indicates INSTALLMONINTERFACE {
           switch (IoControlCode) {
              case MON_ACTIVATE:
                 call ImonActivate;
                 break;
              case MON_DEACTIVATE:
                 Call ImonDeactivate;
                 break;
              case MON_GETENTRY:
                 Call ImonGetEntry;
                 break;
           }
        }
        else {
           pass it down using deviceObject and the
           fastio hook for Ioctl
        }
}
```

ImonAddEntry

```
{
        // similar to ImonEnqueueRegEntry and
        // createRegEntry etc.
        // This function adds a IMON_ENTRY node
        // to our list: this list is eventually
        // returned to InstallMon
        create an entry rec from nonPagedPool
        Grab a mutex to modify EntryList
        Add the entry rec to EntryList
        release the mutex
        KeSetEvent for IMON event object
}
```

ImonActivate

```
{
        // similar to ImonActivate and various
        // Filemon funcs called by
        // FilemonFastIoDeviceControl
        // This is called by our own func that
        // handles IOCTLs when a MON_ACTIVATE is
        // sent by InstallMon
        Look at the ProcessIdList to ensure that we
```

```
                are not already active (1st process id).
                If we are, return with an error with that
                process id
                Grab a mutex
                Add a node to ProcessIdList with the
                processId;
                Release the mutex;
                Create the IMON event object;
                HookDrive(sysDrive);
                HookDrive(destDrive);
                HookRegistry();
    }
```

ImonDeactivate

```
    {
                // Same as above except for MON_DEACTIVATE
                If we are already deactivated
                   return with an error;
                If EntryList is not empty and this is not
                   a forced deactivation then
                   return with error;
                Clear and Delete the IMON event object;
                Free the ProcessIdList;
                Free the EntryList;
                UnhookRegistry();
    }
```

ImonGetEntry

```
    {
                // When the InstallMon sends a MON_GET_ENTRY
                // this function is called.
                Grab the mutex to access EntryList
                get next entry and remove from the
                list;
                if no entry then {
                   if (deactivated and first process in
                         the processIdList is dead) {
                      Make a new entry of "end of data"
                         type;
                   }
                   else {
                      there is some problem (should not
                         happen)
                   }
                }
                copy the entry into the OUT buffer
                release the mutex;
```

```
}
```

HookDrive, UnhookDrive

```
{
        // very similar to Filemon's HookDrive
        // Hence not described here
        // Similarly UnhookDrive
        // When a MON_ACTIVATE comes, we need to
        // make sure that all the requests to the
        // system drive or dest drive are filtered
        // through us. e.g. If "C:" is the system
        // drive, HookDrive will register this
        // as the filter driver for all IO for "C:"
        // Similary for UnhookDrive.
}
```

HookRegistry, UnhookRegistry

```
{
        //similar to Installmon's (Un)HookRegistry
        // We need to make sure that all Registry
        // functions are replaced, with our funcs
        // so that these funcs get called whenever
        // a process is trying to access the
        // registry. Our hook funcs will in turn
        // call the real funcs after queueing an
        // entry
}
```

ImonFastIo* routines

```
{
        // These are very similar to FilemonFastIo
        // routines except that we need to intercept
        // only FAST_IO_WRITE,
        // FAST_IO_MDL_WRITE_COMPLETE,
        // FAST_IO_WRITE_COMPRESSED,
        // FAST_IO_WRITE_COMPLETE_COMPRESSED
        Call the lower layer driver;
        if the request was successfully completed {
          Record the details into our entry rec
          and enqueue it similar to how ImonHookFunc
          does it;
        }
}
```

Interesting issues to deal with:

- Make sure that we use non-paged memory as required. e.g. all the nodes of the processIdList and entryList will be from non-paged pool. According to Bob, we do not need to always allocate non-paged memory – for Registry related nodes (i.e. when it is 'R' type IMON_ENTRY node) we can allocate paged memory. However this needs to be checked – since there will be pointers between these nodes, we need to make sure that this will work properly.
- We have to make sure that a 2-phase install works with this: some setups ask you to reboot the machine and after the reboot the setup continues. For the FSRFD we need to make sure that after the reboot the FSRFD is already loaded before the 2nd phase of the install starts. In the FSRFD we may need to make sure that when the "Runonce" registry key is updated (the installer is trying to do a 2-phase install and setting the 2nd phase exe as the value of "Runonce") we capure that info and accordingly co-ordinate with the Installmon to do the right thing. To ensure that this driver is started at the right time on start-up (since the Builder machine is going to be an in-house dedicated machine, it is okay), we need to add to the System registry (under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`) appropriate values for `Start`, `Group` and `Tag` (and possibly others) value names. Actually this is going to be implemented in InstallMon as a user initiated event in which case the user informs the Builder UI (which in turn informs the InstallMon) that a reboot is imminent. In that case the InstallMon can try to find all possible ways in which the setup.exe is achieving this:
    - The "RunOnce" key or one of its other incarnations (e.g. RunOnceService, RunOnceEx etc) has been modified. We need to figure out exactly which one of the "*RunOnce*" can be modified.
    - The setup.exe has actually added itself (or another exe) to the startup folder. If that is the case, we can do the same trick here: replace that entry with an entry pointing to the BuilderUI with the original setup.exe value as an argument to the BuilderUI.
- An issue that hasn't been resolved is any user interaction (and user input) that has taken place during installation that is being monitored. For example, an installation may ask for a port number that it may store in a registry key. The solution suggested is as follows: This has to be a manual process. The Builder user should record all the manual interaction and manual data input that has taken place. He should recreate the same interaction in the custom DLL that the appInstallBlock provides for that app. This custom DLL at eStream app subscribe time can do the same thing that was observed during original application install. Alternatively we can request the ISV's co-operation in doing this. (May be this bullet should be transferred to a different doc such as the InstallMon LLD.).
- There is another unresolved issue about handling h/w or s/w dependent things that the installer does: we will have to handle this case by case basis and any knowledge we gain as a result of this, we should consolidate in the Builder components. e.g. we may notice that some installations may depend on IE4 or IE5 being there. Of course, one of the pre-requisites of the Builder is that it will be run on a pristine machines, so that we capture a maximal installation when it is taking place.

Testing design

Unit testing plans

This will be unit tested using the INSTALLMON program (or its early prototype). The INSTALLMON program sends all the required Ioctls like MON_ACTIVATE, MON_DEACTIVATE and MON_GETENTRY. The INSTALLMON output will be used to check the correctness of the FSRFD. This means the INSTALLMON itself should be assumed to be correct.

In addition to the above, we actually need to write one or more test programs that exercise the FSRFD. These test programs will be run as if they were App Installers i.e. as child processes of the INSTALLMON. The test programs should be written to exercise:

- All file systems (FAT, FAT32, NTFS, HPFS, compressed drives and others), since the FastIo routines need to be tested.
- For each of the file systems:
    - File create (with and without the old file being there)
    - File update (existing file appended as well as modified in the middle)
    - File touch (e.g. get the version of a DLL file) – this is not yet covered by this design
- Registry updates:
    - create a key
    - delete a subkey
    - delete a named value from a key
    - RegReplaceKey (we need to investigate if this is broken down into smaller reg calls).
    - RegRestoreKey (same comment as above applies)
    - RegSetKeySecurity (we failed to address this in the design)
    - RegSetValueEx
    - RegLoadKey and RegUnLoadKey

Stress testing plans

The FSRFD will be stress-tested using the above testing strategy by varying the rate at which files/registry are updated and under a variety of conditions (memory/disk, other processes).

Coverage testing plans

The unit testing above also covers Coverage testing.

Cross-component testing plans

This will be tested with the Installmon program which is enough for cross-component testing.

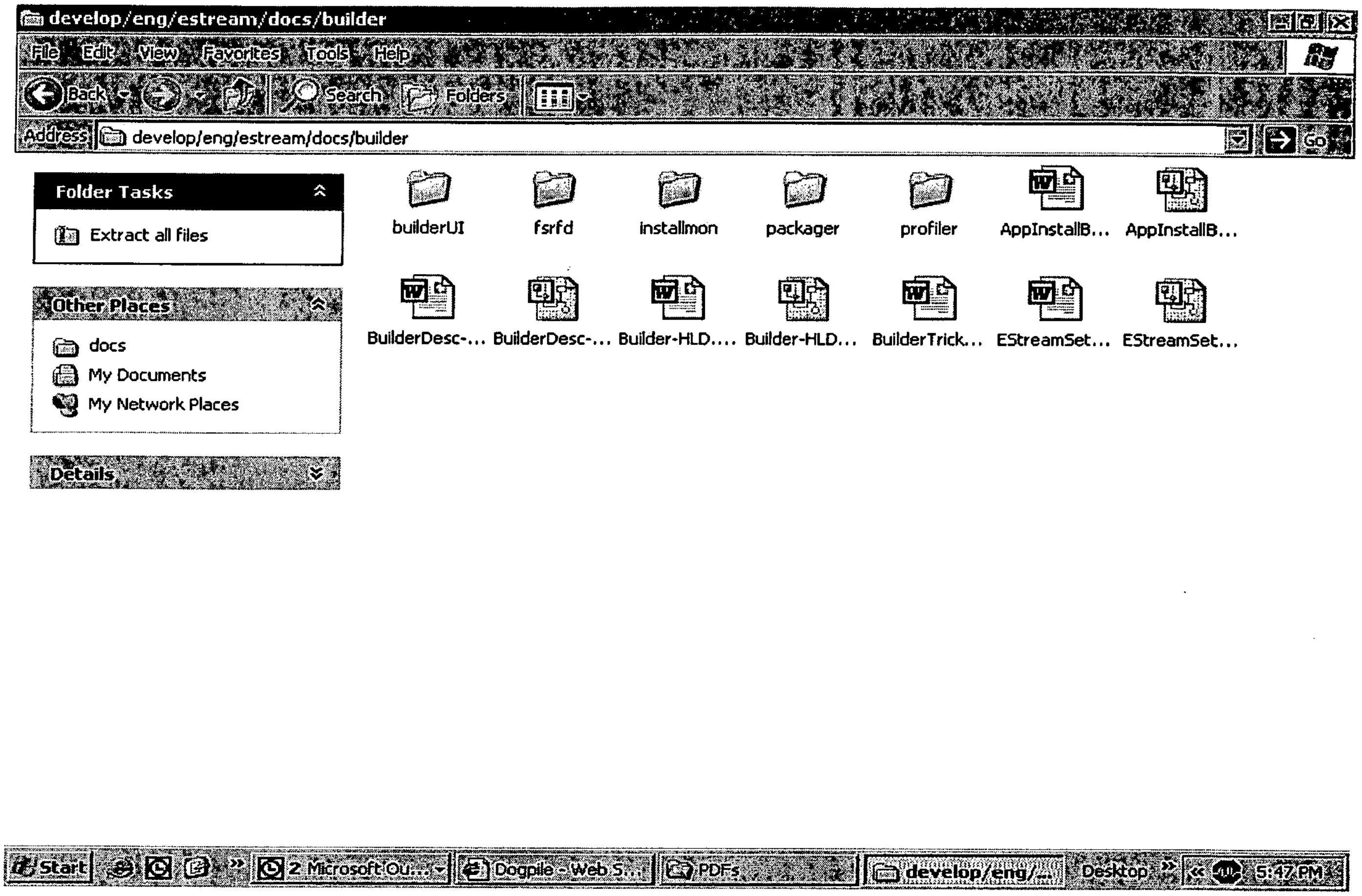
Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

These issues are not necessarily FSRFD related, but are listed here just to remind ourselves.

- Do we need to worry about environment variables? It looks like most installations (and their apps) won't be looking at environment variables.
- What about the .lnk files (shortcuts). It looks like the client guys will have to change all the .lnk files based on the actual client's settings. This issue has been addressed either in the InstallMon or Packager. Pls see those documents. (The `IShellLink` interface has been suggested).
- Also what about when device drivers are installed? There is no impact on the builder but the client will need to reboot and hopefully the existing appInstall-Block and the custom initialization dll should be able to take care of it.


develop/eng/estream/docs/builder
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/builder
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD....
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...
Start
2 Microsoft Ou...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
5:47 PM

[/STANDARD.01]

eStream Installmon Low Level Design

Sanjay M Pujare
<Date>

Functionality

The Installmon is a part of the Builder module that talks to the FSRFD to monitor file system and registry updates initiated by the Builder process. The FSRFD driver just intercepts such requests and records them and returns the recorded data to Installmon when requested by the latter. All the intelligence, such as any decision-making logic, resides in the Installmon.

Note:

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**

2. **Somewhere in the docs (may be the user docs?) it should be mentioned that the Builder's (or rather Installmon's) job could be made easier by the user by following these guidelines:**

   - **Make sure that values in all the registry keys are as distinct as possible. We may need to create a special Win2K or WinNT installation where each key (or ValueName within a key) will be created with a distinct or unique value as far as possible. E.g. If the default Windows installation creates 2 keys FOO and BAR and stores the same value "C:\Windows\System32" in both of them, we wouldn't know which one of those keys is used when a file is copied to "C:\Windows\System32". To solve this problem, all the effort should be made to ensure that FOO and BAR have distinct values.**

   - **When the Builder operator is installing an app under the Installmon, she should also make sure that the install script is given a distinct or unique value for each of the user inputs that may be used to set a registry key or an environment variable. This is especially necessary when the inputs seem to be totally unrelated. For example, vendor name and installation directory name. If both are entered as "Microsoft" then that could cause confusion to the Installmon.**

3. **There are 2 ways in which registry changes and file system changes can be captured:**

   - **using a kernel mode driver such as the FSRFD, Or**

   - **using a diff'ing mechanism for both the registry as well as the file system.**

In this design, we use both the approaches, and record any differences due to these two approaches. We give the user a chance to manually edit the registry/file changes.

Data type definitions

Interface definitions

All of the interfaces used to communicate with the FSRFD are described in the FSRFD LLD document. This LLD document will cover only interfaces with the other modules.

Interfaces with the Builder UI

The public methods of the InstallMon class described below are interfaces to the Builder UI. Specifically these are:

```
void InstallMon::startCapture(PUNICODE_STRING setup_exe,
  PUNICODE_STRING dest, bool upg, FileTable_t *fTbl);
```

This function is called when the user chooses to start monitoring an install. This function is called after the user has entered all the necessary data such as the setup.exe path etc.

```
void InstallMon::stopCapture()
```

This is called in rare cases when the user wants to terminate a running install in abnormal cases.

```
bool InstallMon::checkSetupStatus()
```

This is used by the UI to check the status of the install: whether the file list is ready to be processed and the registry list is ready to be processed etc.

```
void InstallMon::getRegistryList()
```

This function is used to finally get the list of registry changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

```
void InstallMon::getFilesList()
```

This function is used to finally get the list of file system changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

```
void InstallMon::machineToBeRebooted()
```

When the app install program is asking the user to reboot the machine (in the middle of the install) to continue installation, the user has to inform the Builder UI that a reboot is imminent. The InstallMon does the necessary bookkeeping to make sure that monitoring continues after the reboot.

```
void InstallMon::startCaptureAfterReboot(setup_exe, dest,
upg)
```

When the installation continues after the reboot, the Builder UI is automatically invoked, and it calls this function to inform the InstallMon to continue monitoring.

Access DB interface

We will be using an Access DB (or alternately the MSI databases as suggested by Bhaven) to store intermediate results of the Installmon process. There will be 2 tables used: Registry and Files.

Registry table

```
Fullpath :string;         // this is the full path of the key
ValueName :string;        // this is the value name: for
             // (default) use NULL
UpdateType: char;         // Add/update or delete, also
             // 'R' for removed
// combined (Fullpath, ValueName) should be unique i.e.
// primary key
```

Files table

```
Fullpath : string;
UpdateType : char;            // add, update
Kind: char;                   // 'C'opied, 'S'poofed, 'E'stream
FileId: Number;
// Combined (Fullpath, UpdateType, FileId) should be unique
```

RegistryDiff table

```
Fullpath :string;
ValueName:string;
ValueType: char;
Value: something that can store all REG_* types
status: char;   // 'O'riginal, 'C'(add/change), 'D'
// combined (Fullpath, ValueName) should be unique
```

FilesDiff table

```
Fullpath: string;
Type: char;    // file 'F' or dir 'D' etc
Status: char;     // 'O'riginal, ...'C', 'A', 'D' etc
// (Fullpath
```

Component Design

```
struct FileTable_t {
  PUNICODE_STRING name;
  Char type; // spoofed, copied or eFS
  Int  fileId;
};

class InstallMon {
```

```
PUNICODE_STRING setup, destDrive;
bool interruptThread;
EventObject signalMonitor, signalSetup;
bool rebootReq = false;
bool afterReboot = false;
bool completed;
int exitCode;
bool upgrade;
someType envVars;
// TODO: combine the above 2 into an enum
int upgradeFileIdBegin = -1;
int currFileId;
static threadSetup(InstallMon iMon) {
  Remove all the environment variables except
  the ones in iMon->envVars;
  Start the Setup program;
  And wait for it to finish;
  iMon->exitCode = exit code from the process;
  when finished signal the signalSetup event;
}
static threadMonitor(InstallMon *iMon) {
    // this is the func that runs as a separate
    // thread, until we are interrupted or we
    // notice that the setup process has exited.
  get the current
  process id, system drive (using
  GetSystemWindowsDirectory), destDrive and send
  the MON_ACTIVATE message>
  Start threadSetup thread with setup_exe;
  processEnded = false;
  while (!interruptThread) {
      if (rebootReq) {
        only poll for Installmonevent;
        if (signal not set) {
          break from the while loop;
        }
      }
      else {
        WaitForMultipleObjects -> signalMonitor,
        Installmonevent and signalSetup;
      }
      switch (signal) {
      case Installmon event:
        get the MonitorEntry_t record;
        switch (regOrFile) {
        case 'R':
          switch (updateType) {
```

```
          case 'A':
          case 'U':
             add or update (KeyName, ValueName,
                 'N') to
                registry table;
             break;
          case 'D':
             add (KeyName, ValueName, 'D') to
                registry table;
       }
       break;
    case 'F':
       switch (updateType) {
          case 'A':
             // file creation
             add (fullpath, 'A', '?',
                iMon->currFileId++) to files
                table;
             break;
          case 'U':
             add (fullpath, 'U') to files
                table;
             break;
       }
       break;
    case 'E':
       no more data to add;
       if (!processEnded) {
          // something wrong!!!!
       }
       break out of the while loop
       break;
    }
    break;
 case setupevent:
    processEnded = true;
    Send MON_DEACTIVATE to the FSRFD;
    break;
 case signalMonitor event:
    if (rebootReq) {
       kill the threadSetup thread
       clear the signalMonitor event;
    }
    else {
       // TBD?
    }
    break;
```

```
        } // switch
    } // while
    if (processEnded) {
        // normal setup process completion
        // registry and files tables should have
        // all the captured data from the FSRFD
        iMon->diffCapture();
        iMon->completed = true;
    }
} // threadMonitor
void commonCapture(setup_exe, dest) {
     setup = setup_exe;
     destDrive = dest;
     Start the threadFunc thread, and pass 'this' to
     it;
}
void diffCapture() {
   // Bhaven suggested that we could instead do an export
   // from regedit and do a text diff for registry diffs.
   // similarly: It seems the regular Unix diff tool also
   // compares directories. I don't know if it is
   // recursive. If it is, we might be able to use it.
   // Further investigation is recommended for doing
   // files diff.
   Query the OTI\BUILDERSTART key and get the value
   in builderStartTime and delete the
   OTI\BUILDERSTART key;
   // process the registry
   // step 1: query RegistryDiff table
   for each row in RegistryDiff table {
      query the corresponding key+valueName in the
      Windows registry
      if (exists) {
         if (no change to value) {
            update the row status to 'U' for unchanged
         }
         else {
            update the row status to 'C' (changed)
         }
      }
      else {
         update the row status to 'D' (deleted)
      }
   }
   // step 2: enumerate the whole Windows registry
   for each enumerated registry key+value {
      query the RegistryDiff table;
```

```
      if (a row exists) {
         if (status is 'U') {
            delete the row
         }
         else {
            status should be 'C' and values should be
            different bet table and actual registry
            or else display fatal error(?)
         }
      }
      else {
         // no previous value
         add a row to RegistryDiff with (fullpath,
            valueName, ValueType, Value, 'A') to mark
         it as an added registry key
      }
   }
   // now repopulate the FilesDiff table
   // step 1: query FilesDiff table
   for each row in FilesDiff table {
      query the corresponding Fullpath in the actual
      file system;
      if (exists) {
         if (no change (i.e. timestamp before
            builderStartTime)) {
            update the row status to 'U' for unchanged
         }
         else {
            update the row status to 'C' (changed)
         }
      }
      else {
         update the row status to 'D' (deleted)
      }
   }
   // step 2: traverse the whole file system
   for each file/dir in {systemDrive, destDrive} {
      query the FilesDiff table;
      if (a row exists) {
         if (status is 'U') {
            delete the row
         }
         else {
            status should be 'C' and the file/dir
            timestamp should be after builderStartTime
            or else display fatal error(?)
         }
```

```
        }
        else {
          // no previous value
          add a row to FilesDiff with (fullpath,
            'F' or 'D', 'A') to mark it as an added
          file/dir;
        }
      }
    }
    void recursiveFileGetter(curDir) {
      add a row to FilesDiff as (curDir, 'D', 'O');
      for (each element of curDir) {
        if (element is a dir) {
          recursiveFileGetter(element);
        }
        else { // has to be a file(?)
          add a row to FilesDiff as (element, 'F',
               'O');
        }
      }
    }
  public:
    InstallMon(?);
    void setEnvVars() {
      allow the user to set environment variables
      in an interactive way;
      Get the values in envVars;
    }
    void startCapture(PUNICODE_STRING setup_exe,
        PUNICODE_STRING dest, bool upg, FileTable_t *fTbl) {
      clear the Access DB registry, files,
        registryDiff and FilesDiff tables;
      if (upg) {
        populate the files table with data from the
          fTbl array;
        upgradeFileIdBegin = last file id + 1;
        currFileId = upgradeFileIdBegin;
      }
      else {
        currFileId = 0 or 1 (depends on where to start);
      }
      Query the whole registry and
      for (each key and valueName pair) {
        add (key, valueName, valueType, value, 'O') to
          registryDiff;
      }
      Get the current time and store it in some format
```

```
    in a registry key OTI\BUILDERSTART;
    for (curDrive in {systemDrive, dest}) do
    {
      Root = root of curDrive (e.g. "C:\\");
      recursiveFileGetter(Root);
    }
    interruptThread = false;
    afterReboot = false;
    commonCapture(setup_exe, dest, upg);
}
void stopCapture() {
      // abnormal capture termination
      // TBD
      // also there might be normal cases where this
      // func is called when the setup process exit
      // is not detected for some reason (or doesn't
      // happen)! May be we don't have to worry about
      // these things.
}
bool checkSetupStatus() {
    if (!completed) {
      display error and return false;
    }
    if (exitCode is not okay) {
      display error and return false;
    }
}
void getRegistryList() {
    checkSetupStatus() and conditionally return;
    Using appropriate SQL, show (fullpath, ValueName)
      tuples that exist in Registry but not in
      RegistryDiff;
    Tell the user that these were captured by FSRFD
    (installmon) driver but not by the diff process;
    If user chooses to remove any tuples from the
    shown list, mark these with status as 'R' in the
    Registry table;
    Using appropriate SQL, show (fullpath, ValueName)
      tuples that exist in RegistryDiff but not in
      Registry;
    Tell the user that these were captured by diff
    but not by the FSRFD (installmon) driver;
    If user chooses to add any tuples from the
    shown list, add these tuples to the Registry
    table with status copied from RegistryDiff;
    Remove all the rows from the Registry table
    where status is 'R';
```

```
    Now we have all the correct entries in Registry;
    Read each row of Registry and into an array to be
    passed to the caller (most probably the caller of this
    func would have passed an array in which we should
    pass these rows);
}
void getFilesList() {
    checkSetupStatus() and conditionally return;
    Using appropriate SQL, show (fullpath)
      tuples that exist in Files but not in
      FilesDiff;
    Tell the user that these were captured by FSRFD
    (installmon) driver but not by the diff process;
    If user chooses to remove any tuples from the
    shown list, mark these with status as 'R' in the
    Files table;
    Using appropriate SQL, show (fullpath)
      tuples that exist in FilesDiff but not in
      Files;
    Tell the user that these were captured by diff
    but not by the FSRFD (installmon) driver;
    If user chooses to add any tuples from the
    shown list, add these tuples to the Files
    table with status copied from FilesDiff;
    Remove all the rows from the Files table
    where status is 'R';
    Now we have all the correct entries in Files;

    Using appropriate SQL, select files (and not
    dirs) where the file has only 'U' and not 'A':
    this means the setup modified an existing file
    and we cannot handle this; so if this happens
    show a fatal error;

    Read each row of Files and classify it into 'C',
    'S' or 'E' based on the following logic:
    If the file belongs to the app install directory
    (or app drive?) it is an 'E';
    If the file is smaller than a threshold then
    it is 'C' or else it is 'S';

    if (upg) {
      we need to create new file-ids for directories
      that contain new files or directories
      int prevStart = upgradeFileIdBegin;
      int prevEnd = currFileId;
      while ((prevEnd - 1) > prevStart) {
```

```
      using appropriate SQL, select rows from the files
      table where fileId >= prevStart and kind is not
         'C';
      for each such row {
         fullpath = fullpath in the row; (e.g. "C:\A\B")
         dir = parent directory of fullpath (e.g.
               "C:\A")
         select in files a row where fullpath = dir and
         fileId >= upgradeFileIdBegin;
         if (doesn't exist) {
            add in Files a row with (dir, 'X', '?',
               currFileId++);
            // Here 'X' stands for "new file id for a
            // directory created because one of the
            // children has a new file id
         }
      }
      prevStart = prevEnd;
      prevEnd = currFileId;
   }
}
```

Also for each file, change the absolute path to an envVar relative or registry-value relative one: this is a non-trivial task. The way to do this is proposed below. However we need to refine this algorithm based on actual Builder runs on real apps:

For a basic Win2K (or WinNT as the case may be) system create a list of registry keys (and env vars) whose values are normally used as destination paths for copying various kinds of files. To this list also add this app's dest dir as one of the values. Also add all the registry keys/values added as part of this app's install. Create an access table that looks like:

```
Key: String;     // registry or env or other name
Type: char;      // 'R'egistry, 'E'nv, 'D'est dir etc.
Source: char;    // 'S'ystem, 'A'pp, 'U'nknown?
Value: string;
```

Now all of the 'E' files should be relative to dest dir (i.e. 'D' type above). Also any sub-directory strings should either be hard-coded or based on some registry key values where the registry key is of source 'A' (app) above.

```
    All the 'C' and 'S' files should be relative to one of
    the registry keys of source 'S' above and preferably
    type 'R' (since environment variables are not used un-
    der Windows?).

    If there is a sacred file set (i.e. files that are
    so "sacred" for eStream apps, that they were installed
    on the client when the eStream client was installed),
    we need to remove those files from this list;

    Set these values in the table and read the whole
    table in an array (most probably the caller of
    this func would have passed an array in which we
    should pass these rows);
}
void machineToBeRebooted() {
     //The setup is going to reboot the machine.
     //Note that there is a race condition possible here.
     // Suppose the setup.exe has displayed a dialog
     // asking the user to confirm a reboot. At this time
     // the setup.exe has still not written to "Runonce".
     // Only after the user has confirmed (by pressing
     // Okay) will the program write to "runonce" and
     // auto reboot. In that case, this function will be
     // called at the wrong time. We may need to modify
     // the FSRFD to detect changes to "Runonce".
     rebootReq = true;
     signal the signalMonitor event;
     wait for the threadMonitor thread to end;
     Query either our Access database or the actual
     registry to see if Runonce key is set/updated.
     if (not) {
        we cannot handle this reboot situation;
        give fatal error;
        // may be we can look at the
        // Start\Programs\Starup folder and do the
        // same thing we did with Runonce key
     }
     OldRunonce = Old value of Runonce key;
     Set the new value of Runonce key as our Builder
     name (or whichever EXE has the installmon code)
     followed by OldRunonce as the argument to that
     EXE and the destDrive as the next argument and
     upgrade as the next arg;
     Prompt the user to go ahead and say Okay to
     Setup's dialog warning that it is going to
     reboot;
```

```
    }
    void startCaptureAfterReboot(setup_exe, dest, upg) {
        afterReboot = true;
        commonCapture(setup_exe, dest, upg);
    }
};
```

Interesting issues to deal with:

Testing design

Unit testing plans

The unit testing of installmon will be done in conjunction with the FSRFD unit testing. This has been described in the FSRFD-LLD document. In addition, we will be using the sysdiff tool to validate the results of the Installmon.

Stress testing plans

All of the 14 or so applications (that OTI is planning to convert to eStream) installs will be tested under the installmon. Any issues found will be used to fix and improve the installmon functionality.

Coverage testing plans

The testing of the Builder/installmon on the 14 or so app installs should give us enough coverage. Considering that Builder/installmon will be used in-house for some time makes some of the testing issues less significant.

Cross-component testing plans

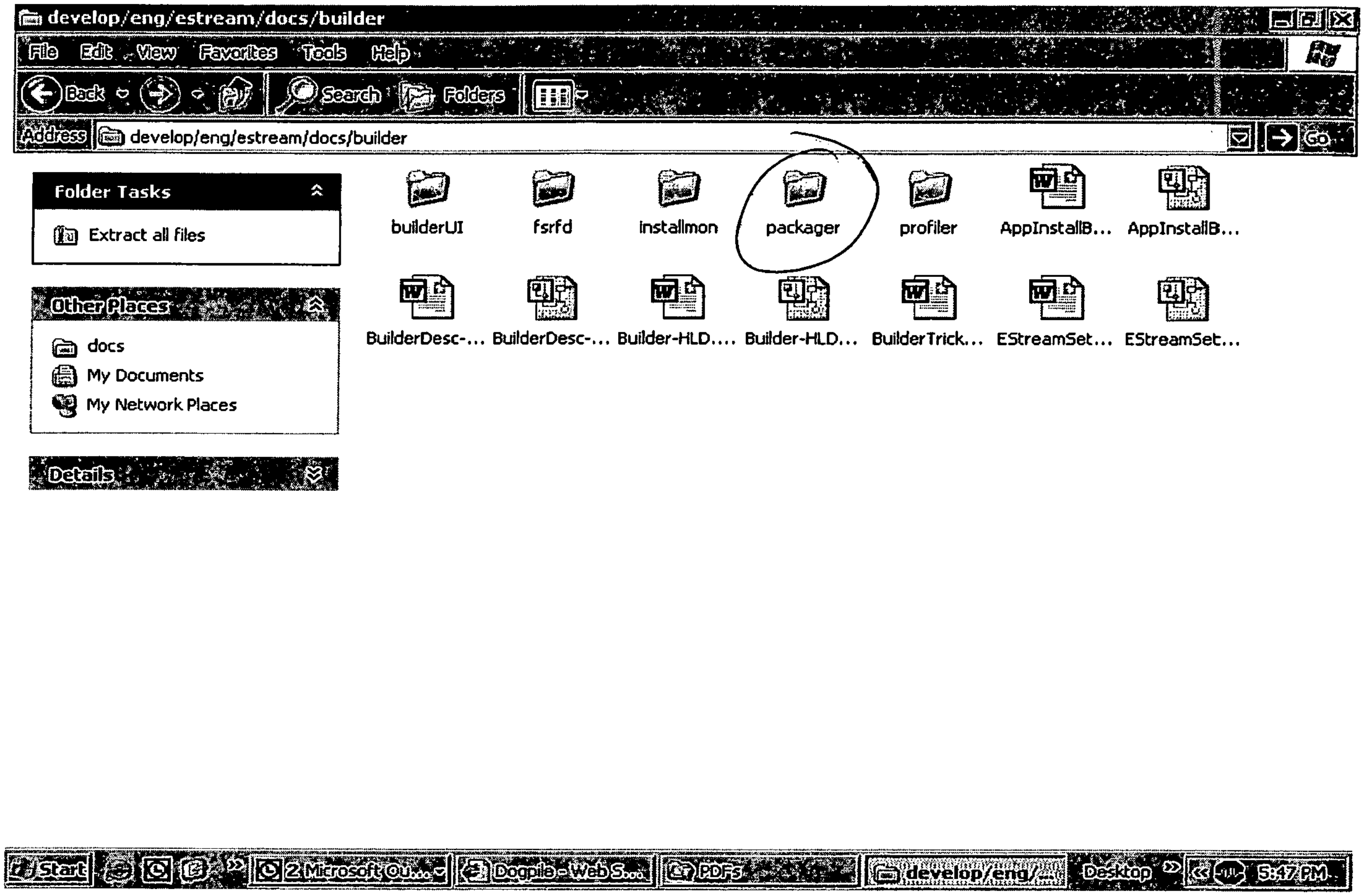
Will be tested as a component of the whole builder.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

- On one of the newgroups someone mentioned a key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs] that we can look at for the shared DLLs that cannot be installed. Need to investigate.
- 2-phase install: some installs need a reboot after which they continue. The key to look at is Runonce (under the same path as SharedDLLs I think).



[/STANDARD.01] [redacted]

eStream Builder Package Manager Low Level Design

Sanjay Pujare and David Lin
Version 0.1
[redacted]

Functionality

The eStream Application Builder Package Manager is responsible for packaging data gathered from the Installation Monitor, the Profile Manager, and the Upgrade Monitor into a set of data called the eStream Set. For the detail format of the eStream Set, see the separate document on eStream Set.

The Package Manager must perform the following task:

- Create the appInstallBlock containing C-File and Registry data from the Install Monitor; Prefetch data from the Profile Manager; and Updated C-File and Updated Registry data from the Upgrade Monitor
- Create a custom installation DLL needed by a specific applications and add to the appInstallBlock
- Create directory files associated with each directory of the application director and add metadata to the directory
- Create directory files associated with each Windows directory containing both the Spoofed files and Z-files
- Create Concatenated Application File (CAF) which is just a juxtaposition of the application files, eStream directory files, and AppInstallBlock
- Create Size Offset File Table (SOFT) which is a mapping of fileNumber to offset of the start of the CAF file
- Create Root Version Table (RVT) which is a mapping from the version of root to the file number of the root directory file
- Archive the CAF, SOFT, and RVT into a single structure called eStream Set suitable for uploading to the eStream Servers.

Data type definitions

The Package Manager doesn't have any internal data types. It must accept and understand data structures received from the Install Monitor and the Profile Manager. See Install Monitor and Profile Manager components for the description of the data structures.

The Install Monitor is responsible for generating the following list of information: list of copied-files, list of spoof-files, list of files with file numbers, list of add registry entries, and list of delete registry entries. The list of copied-files contains the files copied into

non-application specific directories. The list of spoof-files consists of the files too large to be downloaded to the client in the AppInstallBlock. Those files are copied into some special directory on the Z drive for streaming. The list of files with file numbers consists of the files copied into the standard "Program Files" directory and the files that will be spoofed. The registry information is a list of registry key added or removed during the installation of the application.

```
Struct FileIndexTable {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG FileNumber;
   } Entries[NumEntries];
};
Struct FileCopied {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
   } Entries[NumEntries];
}
Struct FileSpoofed {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING OldFilePathName;
      PUNICODE_STRING NewFilePathName;
   } Entries[NumEntries];
};
Struct RegistryInfo {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING KeyName;
      PUNICODE_STRING ValueName;
      PVALUE_DATA ValueData;
   } Entries[NumEntries];
};
Struct IniInfo {
   UINT NumFiles;
   Struct FileEntry {
      PUNICODE_STRING FilePathName;
      UINT NumSections;
      Struct SectionEntry {
         PUNICODE_STRING SectionName;
         UINT NumValues;
         Struct Entry {
            PUNICODE_STRING ValueName;
            PVALUE_DATA ValueData;
         } Entries[NumValues];
      } Entries[NumSections];
```

```
    } Entries[NumFiles];
};
```

The Profile Manager generates AccessCounts and the PrefetchBlocks data with the structures shown below.

```
Struct AccessCounts {
    UINT NumEntries;
    Struct Entry {
        PUNICODE_STRING FilePathName;
        ULONG Frequency;
    } Entries[NumEntries];
};
Struct PrefetchBlocks {
    UINT NumEntries;
    Struct Entry {
        PUNICODE_STRING FilePathName;
        ULONG BlockNumber;
    } Entries[NumEntries];
};
```

The eStream Set has the following data structure (described in more detail in the separate eStream Set document):

```
Struct eStreamSet {
    Struct eStreamSetHeader header;
    Struct eStreamSetRVT rvt;
    Struct eStreamSetSOFT soft;
    Struct eStreamSetCAF caf;
};
```

Interface definitions

Function 1 : CreateEStreamSet

```
// Create the initial eStream Set from the data
// retrieved from the Install Monitor and the
// Profile Manager.
// This function is called only by the Builder
// UI after data is obtained from Install
// Monitor and Profile Manager.
int CreateEStreamSet(
    IN PFILE_INDEX_TABLE FIT,
    IN PFILE_SPOOFED SpoofFiles,
    IN PFILE_COPIED CopiedFiles,
    IN PREGISTRY_INFO AddRegistry,
    IN PREGISTRY_INFO RemoveRegistry,
    IN PINI_INFO IniInfo,
    IN PACCESS_COUNTS AccessCounts,
    IN PPREFETCH_BLOCKS PrefetchBlocks,
```

```
    IN PVOID DllCode,
    IN PUNICODE_STRING Comment,
    OUT PESTREAM_SET EstreamSet)

Input:
    FIT: File Index Tree contains the file
       number of the directories, spoofed
       files, and standard files

    CopiedFiles: pointer to a list of files
       To be copied to AppInstallBlock

    SpoofFiles: pointer to a list of files
       To be spoofed on the client

    AddRegistry: pointer to a list of registry
       Data to add
    RemoveRegistry: pointer to a list of
       Registry data to remove
    IniInfo: pointer to a list of ini changes

    AccessCounts: pointer to the list of
       Files with the access frequency

    PrefetchBlocks: pointer to the prefetch data
       To be inserted into the appInstallBlock
       Of the eStream Set

    DllCode: pointer to DLL Code

    Comment: pointer to comment string

Output:
    EstreamSet: pointer to the eStream Set

Return Value:
    Success or failure of the packaging process

Comments:
    The eStream Set will be large for most
    application.  Intermediate data will be
    stored on the local hard-drive.

Errors:
    OutOfStorage: failure to find enough storage
       For this eStream Set
```

```
FileNotFound: failure to find the files
   Specified by either ListCFiles or
   ListZFiles
```

Function 2 : UpgradeEStreamSet

```
// Upgrade the eStream Set to the latest
// version. This function is only called by
// the Upgrade Manager within the same process.

int UpgradeEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE UpgFIT,
   IN PFILE_SPOOFED UpgSpoofFiles,
   IN PFILE_COPIED UpgCopiedFiles,
   IN PREGISTRY_INFO UpgAddRegistry,
   IN PREGISTRY_INFO UpgRemoveRegistry,
   IN PACCESS_COUNTS UpgAccessCounts,
   IN PPREFETCH_BLOCKS UpgPrefetchBlocks,
   IN PVOID UpgDllCode,
   IN PUNICODE_STRING UpgComment)

Input:
   UpgFIT: File Index Tree contains the file
      number of the directories, spoofed
      files, and standard files

   UpgCopiedFiles: pointer to a list of files
      To be copied to AppInstallBlock

   UpgSpoofFiles: pointer to a list of files
      To be spoofed on the client

   UpgAddRegistry: pointer to a list of
      Registry data to add

   UpgRemoveRegistry: pointer to a list of
      Registry data to remove

   UpgAccessCounts: pointer to the list of
      Files with the access frequency

   UpgPrefetchBlocks: pointer to the prefetch
      Data to be inserted into the
      AppInstallBlock Of the eStream Set

   UpgDllCode: pointer to DLL Code
```

```
   UpgComment: pointer to comment string

Output:
   EstreamSet: pointer to the eStream Set

Return Value:
   Success or failure of the packaging process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
   OutOfStorage: failure to find enough storage
     For this eStream Set

   FileNotFound: failure to find the files
     Specified by either ListCFiles or
     ListZFiles
```

Function 3 : ModifyEStreamSet

```
// Insert new data into different sections
// of estream set.  This function is overloaded
// to handle different section data

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE FIT,
   IN PPREFETCH_BLOCKS PrefetchBlocks)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PREGISTRY_INFO AddRegistry,
   IN PREGISTRY_INFO RemoveRegistry)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE FIT,
   IN PFILE_SPOOFED SpoofFiles,
   IN PFILE_COPIED CopiedFiles)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PVOID DllCode)

int ModifyEStreamSet(
```

```
   INOUT PESTREAM_SET EstreamSet,
   IN PUNICODE_STRING Comment)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PUNICODE_STRING LicenseText)

Input:
   FIT: File Index Tree contains the file
      number of the directories, spoofed
      files, and standard files

   CopiedFiles: pointer to a list of files
      To be copied to AppInstallBlock

   SpoofFiles: pointer to a list of files
      To be spoofed on the client

   AddRegistry: pointer to a list of registry
      Data to add
   RemoveRegistry: pointer to a list of
      Registry data to remove
   IniInfo: pointer to a list of ini changes

   AccessCounts: pointer to the list of
      Files with the access frequency

   PrefetchBlocks: pointer to the prefetch data
      To be inserted into the appInstallBlock
      Of the eStream Set

   DllCode: pointer to DLL Code

   Comment: pointer to comment string

Output:
   EstreamSet: pointer to the new eStream Set

Return Value:
   Success or failure of the insertion process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
```

```
OutOfStorage: failure to find enough storage
   For this eStream Set

FileNotFound: failure to find the files
   Associated with the prefetch blocks
```

Component design

The pseudo-code for the function *CreateEStreamSet* is described below:

{
    Create AppInstallBlock (AIB) from the following input files:
    - SpoofFiles
    - CopiedFiles
    - AddRegistry
    - RemoveRegistry
    - Prefetch
    - Comment
    - DLLcode

    Assign AppInstallBlock with a unique fileNumber given by the IM;
    Record Root fileNumber in the first entry of Root fileNumber Table (RFT);
    Move AppInstallBlock under the Root directory by adding a new entry in the
        Directory structure;
    Create a Concatenation Application File (CAF) header;
    Create a Size Offset File Table (SOFT) header;
    For each (file in FIT) {
        If (file is a directory) {
            Create the directory with new list of fileNumber, filename, and
                Metadata;
        } Else {
            Find the file in the proper location on the HD;
        }
        Append the file or directory to the end of the CAF file;
        Append the fileNumber, offset into CAF, and size of file in SOFT;
    }
    Archive CAF, SOFT, and RFT into a single eStream Set;
    Return eStream Set;
}

The pseudo-code for the function *UpgradeEStreamSet* is mentioned below:

```
{
    Extract previous version PrevAppInstallBlock from eStream Set;
    Create new AppInstallBlock with new FileNumber;

    Extract PrevSpoofFiles and PrevCopiedFiles from PrevAppInstallBlock;
    Divide the C-Files into SpoofFiles and CopiedFiles;
    Add PrevSpoofFiles to SpoofFiles;
    Add PrevCopiedFiles to CopiedFiles;

    Extract PrevAddRegistry and PrevRemoveRegistry data from
        PrevAppInstallBlock;
    Add any unique ((UpgAddRegistry plus PrevAddRegistry) minus
        UpgRemoveRegistry) in the new AppInstallBlock AddRegistry section;
    Add any unique ((UpgRemoveRegistry plus PrevRemoveRegistry) minus
        UpgAddRegistry in the new AppInstallBlock;

    Add UpgPrefetch data to new AppInstallBlock;
    Add UpgDllCode data to new AppInstallBlock;
    Add UpgComment data to new AppInstallBlock;

    For each (directory in UpgFIT) {
        If (any child fileNumber has changed) {
            Create new directory with updated fileNumber;
            Append file to end of Concatination Application File (CAF);
            Append Size Offset File Table (SOFT) with new entry;
        }
    }
    Append new AppInstallBlock to the end of CAF file;

    Prepend Root FileNumber Table (RFT) with new Root entry;

    Archive CAF, SOFT, and RFT into a single eStream Set;
    Return eStream Set;
}
```

The pseudo-code for the overloaded function *ModifyEStreamSet* is mentioned below:

```
{
```

// not needed unless merging of uploaded profile data is supported

}

Testing design

This document must have a discussion of how the component is to be tested.

- **Unit testing plans**

  The plan for unit testing Package Manager includes the development of a driver program. This driver interfaces to the Package Manager and invokes the functions with different parameters. The list of possible cases is described below:

  1. Test all interfaces by driving the input parameters with different type of add and remove registry values.
  2. Test all interfaces by driving the input parameters by varying numbers of spoof and copied files.
  3. Test all interfaces by driving the input parameters with some prefetch information.
  4. Test all interfaces for meaningless input values from the IM and PM.
     - Prefetch block containing file number not assigned by IM.
     - IM assigning non-continguous file numbers.
  5. Test upgrade interface for capability to detect and handle bad eStream Set gracefully.
  6. Test upgrade interface and make sure it can detect overlapping file number assignments.
  7. Test upgrade interface and make sure prefetch blocks are not referencing old file number from previous versions.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  The output data from the Package Manager is called the eStream Set. This eStream Set is the input to a stand-alone test program called the *eStream Extractor*. The Extractor unpacks and 'install' the eStream Set into the local machine without an eStream client file system installed. This test is used to quickly verify that the eStream Set can be run on a pristine machine. Some of the possible variations of the Extractor test includes:

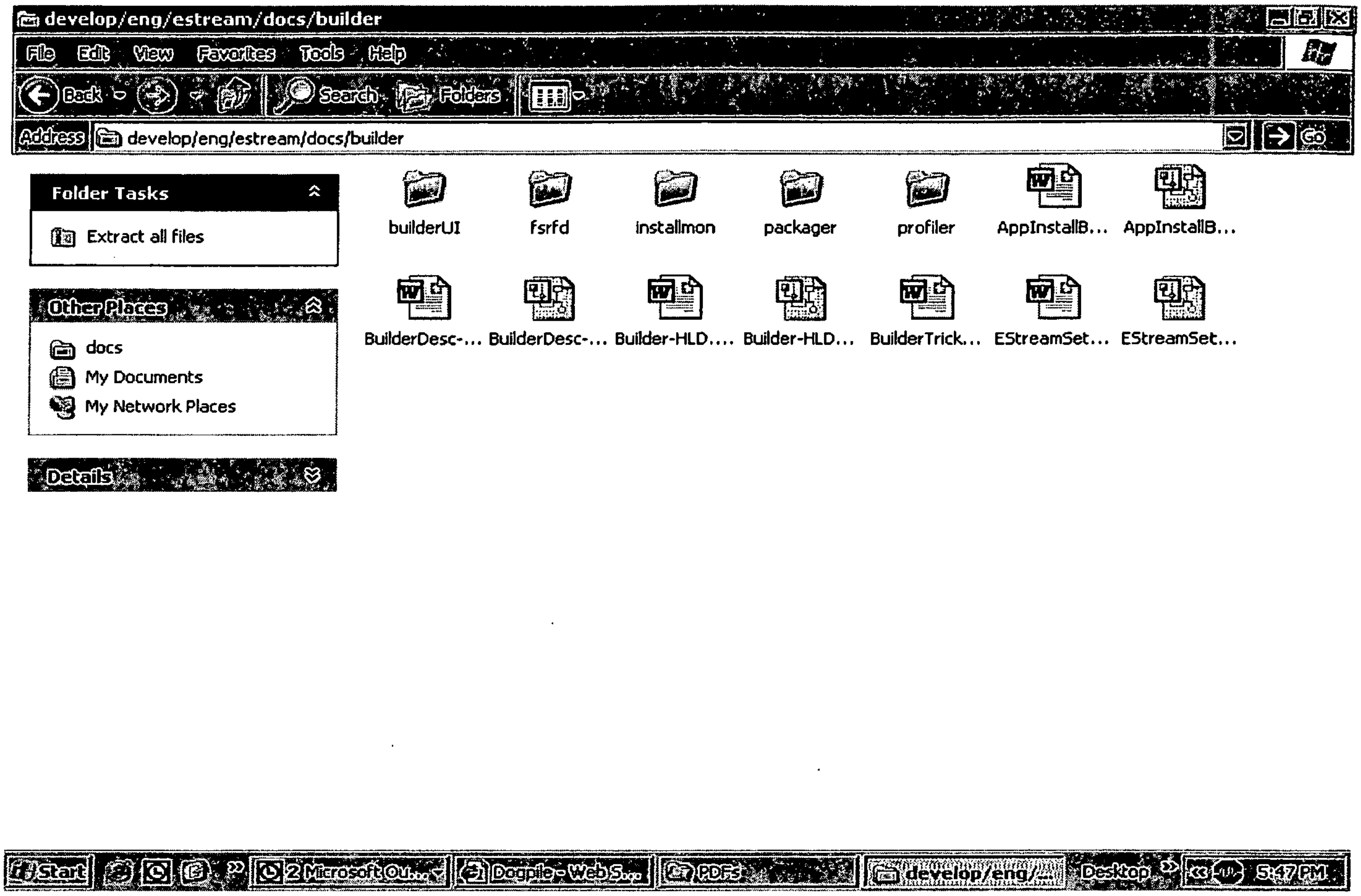
  1. Non-default system variable names. I.e. %SystemRoot% set to "D:\Win" instead of "C:\Winnt".
  2. Non-default eStream FS drive letter. Use Y instead of Z.

Upgrading/Supportability/Deployment design

The Package Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- Which Builder component creates the installation DLL when the application needs the custom installation code? Is a new component needed to create the custom DLL separately and insert into AppInstallBlock in the eStream Set as needed?


develop/eng/estream/docs/builder
File Edit View Favorites Tools Help
Back
Search
Folders
Address
develop/eng/estream/docs/builder
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD....
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...
Start
2 Microsoft Ou...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
5:47 PM

eStream Builder File Access Monitor Low Level Design

Sanjay Pujare and David Lin
Version 0.1
[redacted]

Functionality

The eStream Application Builder File Access Monitor (FAM) is a kernel-mode device driver that behaves as a file filter driver to has the following responsibilities:

- Monitor any running application's request to access a file or directory
- Track application file and directory accesses
- Track file metadata queries
- Start and stop profiling via IOCTL requests from the user-mode program
- Return the file access data to the user-mode program via I/O Request Packet (IRP)
- Return any error conditions to the user-mode program via IRP

The File Access Monitor is based on the 'Filemon' program. The source code for the program is available free for download over the Web at http://www.sysinternals.com/filemon.htm.

Data type definitions

The File Access Monitor (FAM) monitors a sequence of file block accesses by a particular process or one of its child processes. The FAM also tracks any queries on the file metadata. The combination of the file content and metadata is returned to the Profile Manager for further processing. The following is the data structure externally visible to the other subcomponents outside FAM.

```
Struct SequenceData
{
  UINT NumEntries;
  Struct Entry
  {
    PUNICODE_STRING FilePathName;
    BOOL IsAccessingMetadata;
    ULONG Offset;
    ULONG Size;
  } Entries[NumEntries];
};
```

The FileName contains the null terminating string of the file accessed. IsAccessMetadata flag indicates if the access is on the file metadata or the file content. If the operation is on

the file content, then the fields 'Offset' and 'Size' indicate the location of the read or write operations. Otherwise, the fields 'Offset' and 'Size' are not used.

Interface definitions

Function 1 : Hooks into user defined IOCTL calls

```
// The following is a Fast I/O Device Control
// call interface.  Each of the user IOCTL
// call to the driver is described here.
// The OICTL input and output parameters are
// stored on the IRP on the InputBuffer and
// OutputBuffer respectively.
// This function must be called only by NT I/O
// Manager.

BOOLEAN FileSysFastIoDeviceControl(
   IN PFILE_OBJECT FileObject,
   IN BOOLEAN Wait,
   IN PVOID InputBuffer,
   IN ULONG InputBufferLength,
   OUT PVOID OutputBuffer,
   IN ULONG OutputBufferLength,
   IN ULONG IoControlCode,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Input:
   IoControlCode==IOCTL_FAM_VERSION
      OutputBuffer: version number of the driver

   IoControlCode==IOCTL_FAM_START
      InputBuffer: process ID to monitor

   IoControlCode==IOCTL_FAM_STOP
      OutputBuffer: stop profiling

   IoControlCode==IOCTL_FAM_GETDATA
      OutputBuffer: get sequence data

   IoControlCode==IOCTL_FAM_GETSTATUS
      OutputBuffer: get status from driver


Output:
```

```
Comments:
   The IOCTL calls from the user program to
   the device driver is either through the
   dispatcher or through the Fast I/O
   interface.

Errors:
```

Function 2: DriverEntry

```
// Called by the NT system to initialize
// driver.  The following entries are hooks
// into the OS and are not called by any of our
// component directly.
NTSTATUS DriverEntry(
   IN PDRIVER_OBJECT DriverObject,
   IN PUNICODE_STRING RegistryPath)

Comments:
   Initialize the driver
```

Function 3: Hooks into Fast I/O functions

```
// NT Fast I/O calls.  These are some of the
// hooks into the OS
NTSTATUS FileSysFastIoRead(
   IN PDRIVER_OBJECT DriverObject,
   IN PLARGE_INTEGER FileOffset,
   IN ULONG Length,
   IN BOOLEAN Wait,
   IN ULONG LockKey,
   OUT PVOID Buffer,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Comments:
   Hooks into Fast I/O Read
```

Function 4: Hooks into dispatcher functions

```
// Besides hooks into Fast I/O calls, we
// must also hook into each of the major
// functions like IRP_MJ_CREATE, IRP_MJ_READ,
// etc...
FileSysHookRoutine(
   PDEVICE_OBJECT HookDevice,
```

```
    IN IRP Irp)
```

Component design

I/O Hook location

The trickiest part of the File Access Monitor (FAM) component design is determining the locations in the Operating System to hook the routines. FAM must provide driver entry function for initializing the driver. It must also provide hooks into the OS to monitor read and write file operations. In addition, it needs to monitor access to file metadata. And finally, it has to provide the user-mode program a way to communicate to the FAM through the IOCTL calls.

The FAM must behave like any other Windows kernel-mode drivers by exporting the standard *DriverEntry* function. The *DriverEntry* function has the following purposes:

- Check for OS build version.
- Setup the device name
- Call OS routine to create the device
- Make symbolic link to allow device access from Win32 programs
- Create dispatch points for all routines that must be handled
- Setup Fast I/O hooks
- Initialize all data structures

In addition to the *DriverEntry*, the FAM must handle five user defined IOCTL calls.

- IOCTL_FAM_VERSION
- IOCTL_FAM_START
- IOCTL_FAM_STOP
- IOCTL_FAM_GETDATA
- IOCTL_FAM_GETSTATUS

In IOCTL_FAM_START, the handler receives the process ID from the user-mode program. It uses this process ID to filter out relevant file and metadata accesses. In IOCTL_FAM_STOP, the handler stops monitoring and recording any file accesses. In IOCTL_FAM_GETDATA, the handler packages the file access sequence in the I/O Request Packet (IRP) to be returned to the user-mode program. Finally, in IOCTL_FAM_GETSTATUS, the handler returns its current status. This status includes: FAM_STATUS_OK, FAM_STATUS_ERROR, and FAM_STATUS_PROFILING.

In addition to the user defined IOCTL hooks, the FAM must add hook into both the dispatch points and the Fast I/O calls to monitor all read and write requests. In addition, the FAM monitors any metadata accesses. The following is a list of Fast I/O calls it must hook:

- FastIoRead
- FastIoWrite

- FastIoMdlReadComplete
- FastIoMdlWriteComplete
- FastIoReadCompressed
- FastIoWriteCompressed
- FastIoQueryBasicInformation
- FastIoQueryStandardInformation

In the routine to handle FastIoRead and FastIoWrite, the driver must determine the process ID making this request. If the process is in the list of monitoring processes, the file name, file offset, and size is recorded and added to the profile sequence list. In the routine to handle FastIoQueryBasicInformation and FastIoQueryStandardInformation, the driver records the file name associated with this metadata query.

In addition to hooks to the Fast I/O calls, the I/O may call the File System services through standard Windows NT dispatch points. The following is a list of dispatch points to be handled by FAM:

- IRP_MJ_CREATE
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_DIRECTORY_CONTROL + IRP_MN_QUERY_DIRECTORY
- IRP_MJ_QUERY_INFORMATION
- IRP_MJ_SET_INFORMATION
- IRP_MJ_QUERY_EA
- IRP_MJ_SET_EA

The routine to handle IRP_MJ_READ, IRP_MJ_WRITE, and IRP_MN_QUERY_DIRECTORY is handled by the same function as the routine for handling FastIoRead and FastIoWrite. The routine to handle IRP_MJ_QUERY_INFORMATION, IRP_MJ_SET_INFORMATION, IRP_MJ_QUERY_EA, and IRP_MJ_SET_EA are handled by the same function as the routine for handling FastIoQueryInformation.

Communication with user-mode component (Profile Manager)
Besides using the IOCTL to send profile data to the Profile Manager, the FAM must also signal the Profile Manager when new data is available for retrieval. The Profile Manager wakes up from the signal by the FAM and retrieves the information on the blocks of files accessed. FAM also signals the profile manager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWaitForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Process Filtering
The FAM must filter the profile information so only relevant data relating to the application under profiled is obtained. This is accomplished by filtering the data according to

the process ID invoking the file access operations. When the FAM is started, the Profile Manager sends its process ID. FAM assumes all child processes of the Profile Manager process ID is to be monitored since the Profile Manager invokes all applications using *CreateProcess()* API. Thus, the new processes all inherit Profile Manager as its parent process ID. The process filtering is accomplished using *PsSetCreateProcessNotify-Routine()* to add a hook to the OS. FAM is notified whenever there is a new process created. The process ID is recorded in a list if its ancestor is the Profile Manager process ID. This list is used to filter the profile data gathered by FAM.

Locks

Since multiple threads may be entering different sections of FAM and accessing different data structures, appropriate locks must be used to prevent multiple threads from reading and writing at the same time. *ExInitalizeResourceLite(), ExAcquireResourceExclusiveLite(),* and *ExReleaseResourceLite()* are used when shared data structure is accessed. These APIs have the requirement that the kernel APCs must be disabled before calling and that IRQL must be lower than DISPATCH_LEVEL. This can be accomplished by using *KeEnterCriticalRegion()* and *KeLeaveCriticalRegion()*. The following is a sample code using these APIs:

```
ERESOURCE gResource; // global variable

KeEnterCriticalRegion()
ExAcquireResourceExclusiveLite(&gResource, TRUE);

<critical section of code>

ExReleaseResourceLite(&gResource);
KeLeaveCriticalRegion();
```

Testing design

- **Unit testing plans**

  The plan for unit testing of the FAM consists of using the Profile Manager (PM) and a File Access Driver (FAD) as the test drivers. The PM tests user-defined IOCTL calls. The FAD creates desired data pattern from the OS's I/O Manager to the FAM. The FAD tests the FAM's ability to monitor file accesses by querying files and directories in a particular order. Together, the PM and FAD test coverage of the FAM is complete. The following is a list of tests:

  1. Test each user-defined IOCTL interface via PM by sending border cases.
  2. Test to make sure FAM captures every file and directory access via standard file I/O requests from a user-mode program called FAD.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

Other Builder components log error messages to a predefined file. The kernel-mode programs do not have the capability to read/write to a file. Since FAM is a kernel-mode program, an alternative method of reporting error messages has to be developed. Current, the FAM has a user-defined IOCTL interface (IOCTL_FAM_GETSTATUS) to retrieve the error messages. FAM keeps a stack of error messages encountered and reports the stack of error messages at the request by an appropriate user-mode program.

Open Issues

- Exactly which Fast I/O calls need to be hooked to get all the read and write operations for file accesses?
- Along the same line, which dispatch points need to handled to get all the read and write operations for file accesses?
- Have we hooked into all possible places where the metadata accesses can occur?
- Does the FAM need to hook into FileLock and FileUnlock operations?

eStream Builder Profile Manager Low Level Design

Sanjay Pujare and David Lin
Version 0.3
[redacted]

Functionality

The eStream Application Builder Profile Manager is responsible for the following:

- Receive request from the UI Component for one or more application executables to profile.
- Accumulate each run of the profile data in a data structure suitable for merging.
- Invoke each application executable for a fixed amount of time, for a fixed number of prefetch blocks, for a simple start-stop of the program, or for multi-level profiling based on scripts or manual usage of an application.
- Communicate with File Access Monitor (FAM) kernel-mode driver using IOCTLs to start and stop profiling.
- Obtain the complete file access sequence data from the FAM.
- Process the file access sequence into two parts: a table of file access frequency and a list of prefetch blocks.
- Send the resulting profile data to Package Manager component for integration into the AppInstallBlock.

This component will probably exist as a class object and will be instantiated by the Builder user interface component. The component will run in the same process as the user interface component. Please see Builder User Interface component document for more information on that component.

Data type definitions

The Profile Manager imports the file access sequence from the FAM. The data structure is described below: (Please see the File Access Monitor for detail information on the meaning of each field in the data structure)

```
Struct SequenceData
{
   UINT NumEntries;
   Struct Entry
   {
      UNICODE_STRING FilePathName;
      BOOL IsAccessingMetaData;
      ULONG Offset;
      ULONG Size;
   } Entries[NumEntries];
```

```
};
```

Profile Manager creates two data structure from the data received from the Install Monitor and the FAM. The consumer of the output data structure is the Package Manager. These data structures is described in the following two structures:

```
Struct PrefetchBlockList {
   UINT NumSections;
   Struct PrefetchBlocks {
      UINT BlockType;
      UINT NumEntries;
      Struct Entry {
         UNICODE_STRING FilePathName;
         ULONG BlockNumber;
      } Entries[NumEntries];
   } Entries[NumSections];
};
Struct ProfileApplications {
   UINT NumEntries;
   Struct Entry {
      UNICODE_STRING FilePathName;
      UNICODE_STRING Arguments;
   } Entries[NumEntries];
};
```

The access count is used to order the list of files in a directory according to metadata file access frequency. The prefetch data is encorporated into the AppInstallBlock by the Package Manager to be used by the eStream Client.

Interface definitions

Function 1 : StartProfiling

```
int StartProfiling(
   IN PPROFILE_APPLICATIONS AppList,
   IN UINT Type,
   IN UINT TypeData,
   OUT PPREFETCH_BLOCKS PrefetchBlocks)

Input:
   AppList: a list of file pathnames and
      Arguments to run the application.

   Type: type of profiling to do
      SIMPLE: start and stop application
      TIMEBASED: profile for a fixed time and
```

```
            terminate application
         SIZEBASED: profile for a fixed size of
            Profile data

      TypeData: extra data for profile type
         If ProfileType==SIMPLE, TypeData is
            Ignored
         If ProfileType==TIMEBASED, TypeData is
            Length of time in seconds
         If ProfileType==SIZEBASED, TypeData is
            Size of profile data in bytes
         If ProfileType==COMMANDBASED, TypeData is
            Pointer to a possible list of script
            Files to be invoked

      PrefetchBlocks: List of prefetch blocks
         To add to the AppInstallBlock

   Return value:
      Success or failure code of the profiling

   Comments:
      The profile manager component actually send
      IOCTLs to the file filter device driver to
      Start and stop the gathering of the profile
      Data and to retrieve the profile data.

   Errors:
      FileNotFound: some of the application
         Executables in the list may not exist or
         Not readable
      ProfileTimeout: failed to gather the desired
         Size of the profile data after certain
         Amount of time
      DriverFailure: File Access Monitor return
         Failure code to the Profile Manager which
         Must propagate the error to the user
         Interface
```

Component design

Application Invocation

To start profiling, the component must have a list of application pathname to be invoked. The pathname may be an executable with a list of arguments. Or the pathname may be a Windows short-cut file. If the pathname is an exe file, then the standard Win32 API *CreateProcess()* is used. On the other hand, if the file is a Windows short-cut file, then the

component needs to extract relevant information from the short-cut file to make the proper *CreateProcess()* invocation. The following is a pseudo-code for extracting path-name and argument information from the short-cut file using IShellLink interface:

```
GetInfoFromShortCutFile(char *strPath, char *strArg)
{
  IshellLink *psl;
  WIN32_FIND_DATA fd;
  if (SUCCEEDED( CoCreateIstance(CLSID_ShellLink,
                 NULL,
                 CLSCTX_INPROC_SERVER,
                 IID_IShellLink,
                 (LPVOID*) &psl)))
  {
    psl->GetPath(strPath, MAX_LEN, &fd, 0);
    Psl->GetArguments(strArg, MAX_LEN);
    psl->Release();
  }
}
```

Command-based Profiling

One of the options for profiling includes the ability to identify blocks of files accessed when specific application command is invoked. The profiler prompts the user for the desired actions (ie. Open document, save document, etc) and gathers file blocks that are accessed corresponding to those actions. These commands are saved into the AppInstall-Block for eStream client to intelligently pick the proper set of blocks to stream. The following is an enumeration of some divisions of prefetch blocks:

- Start Application
- End Application
- Save Document
- Open Document
- Cut Sections
- Copy Sections
- Paste Sections

In the current design, the profiler divides the code into two prefetch sections. The first section contains the critical blocks necessary for efficient startup of the streamed application for the very first time. The blocks of code in this section include starting of application. The second section is the common blocks necessary for efficient running of the streamed application. This includes common user operations like opening and saving of document.

Communication with kernel-mode driver (FAM)

The Profile Manager communicates with the kernel-mode driver to retrieve the information on the blocks of files accessed. The profile manager waits for a signal from the FAM indicating a new data is available for retrieval. FAM also signals the profile man-

ager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWait-ForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Pseudo-code
The pseudo-code for the function *StartProfiling* is described below:

```
{
    Initialize GlobalFileAccessCounts and GlobalPrefetchBlocks;
    Load FAM if not loaded;
    For each (executable in the list of application) {
        Start FAM by sending it process ID of the Profile Manager;
        Create new process for this executable and run it;
        Switch (Type of Profiling) {
        Case SIMPLE:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until application start up;
            Break;
        Case TIMEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until fixed time unit;
            Break;
        Case SIZEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } while (size < fixed amount);
            Break;
        Case COMMANDBASED:
```

```
            For each command in the list {
                If (script exist)
                    Run script on the executable program;
                Else
                    Prompt operator for proper action;
                Loop {
                    Wait for an event from FAM;
                    Get Status from FAM;
                    Get SequenceData from FAM;
                } Until script completed;
            }
        }

        Send WM_QUIT message to the application process;
        Loop {
            Wait for an event from FAM;
            Get Status from FAM;
            Get SequenceData from FAM;
        } Until application quit;
        Inform FAM to stop gathering profile data;

        Compute PrefetchBlocks from the SequenceData and append to
            GlobalPrefetchBlocks;
    }
    Unload File Access Monitor (if possible);
    Return GlobalPrefetchBlocks;
}
```

Testing design

- **Unit testing plans**

The plan for unit testing the Profile Manager includes developing a driver to connect to the interface between the Profile Manager and the Builder UI. The driver conducts the following types of tests:

1. Test different type of profiling including simple profiling, time-based profiling, size-based profiling, and script-based profiling.

2. Test different executable programs and make sure the output data "makes sense".
3. Test a list of executables for merging capability of the Profile Manager.
4. Test the interface between Profile Manager (PM) and the File Access Monitor (FAM) using FAM as the test driver. The FAM can check for any valid IOCTL calls from the PM. FAM can also reply to IOCTL calls with different values in the IRP to simulate all possible cases.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

The Profile Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- How to automate profiling so the application doesn't require any user intervention? Look into using Rational TestSuite programs.
- Can a subset of Winstone be used for profiling? How do we determine which part of the profile data is more useful to the end-user?
- Should Profile Manager actually create data structures like PrefetchBlocks that require FileNumber assignments? Or should Profile Manager just create the output data without knowning FileNumbers? Then Package Manager associates the file numbers assigned by the Install Monitor with the profile data gathered by the Profiler.


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/client
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
2 Microsoft Ou...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
6:36 PM

[/STANDARD.01]

eStream Application Install Manager Low Level Design

Nicholas Ryan
Version 1.5
[redacted]

Functionality

The Application Install Manager (AIM) is a component of the eStream client executable. It is responsible for installing and uninstalling eStream applications at the request of the License Subscription Manager (LSM). AIM uses the information contained in an AppInstallBlock to prepare the user's system for execution of a given eStream application. It creates registry entries, copies files, and updates the file spoofing database. The user can then launch his application via a local shortcut or a shortcut on the eStream drive. Uninstallation involves undoing all changes made to the user's system by AIM during installation.

Data type definitions

This component uses the AppInstallBlock, but doesn't define it. This is defined in a low-level design document for the Builder component.

The AppInstallBlock is a binary data file with a versioned interface, basically consisting of:

- a header
- a list of files to install or send to the file spoofer
- a list of registry entries to install or remove
- a set of prefetch requests to communicate to the profile/prefetch component
- a set of initial profile data to communicate to the profile/prefetch component (post-version 1.0)
- a comment section
- an embedded DLL that can be loaded and executed for custom install needs
- a section containing a license agreement to be shown to the user

Many of the AIMsc functions take an AIBFileRef as an argument, which is an opaque pointer to a Builder-provided class called 'AppInstallBlock'. This utility class will be shared by the Builder and the AIM code. It encapsulates the complexity of reading from and writing to the various sections of an AppInstallBlock file.

Also, each application has a prefetch data file created for it an install time that is initialized with prefetch data from the AppInstallBlock. This data file is named and located as described in the Component Design section, and consists of a list of critical blocks followed by a list of common blocks. The file begins with the following structure:

```
typedef struct
{
   OTUInt32  numCriticalBlocks;
   OTUInt32  numCommonBlocks;

} PrefetchFileHeader, *pPrefetchFileHeader;
```

The remained of the file consists of a non-padded list of the following structures:

```
typedef struct
{
   OTUInt32  fileNumber;
   OTUInt32  blockNumber;

} PrefetchItem, *pPrefetchItem;
```

The following data types are used in the AIM and AIMsc interfaces:

```
typedef void *AIBFileRef;
```

Error codes are defined in the OTError enumeration, which lives in a file called ot_errors.h.

Interface definitions

Application installation/uninstallation

There are only two functions exposed by AIM, one for application installation, and another for application uninstallation. Only the License Subscription Manager will be calling these functions.

*OTError AIMInstallApplication(OTUInt8 appId[16], const char *pathToAIB)*

Parameters

appId

[in] The application ID of the eStream application to install.

pathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to install.

Return Values

OT_SUCCESS if all the actions specified in the AppInstallBlock were performed successfully, an error code otherwise.

Comments

None.

OTError AIMUninstallApplication(OTUInt8 appId[16])

Parameters

appId

[in] The application ID of an existing eStream application to uninstall.

Return Values

If the specified application ID is not recognized, or the original AppInstallBlock is not found, an error code will be returned. Otherwise, AIM will make an attempt to undo all of the actions it performed while installing this application. It will return OT_SUCCESS if it undid enough of these actions so that any future installation of the same application will succeed.

Comments

None.

AIM Sub-Component Interface

Much of the functionality required by the AIM design will be useful to the Builder testing framework as well. This functionality will be treated as a sub-component within the AIM component, called AIMsc, and will export a well-defined interface. That interface is defined as follows.

*OTError AIMscOpenAppInstallBlock(const char *pathToAIB, AIBFileRef *pAIBFile)*

Parameters

pathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to open.

pAIBFile

[out] Returns a reference to an open AppInstallBlock file.

Return Values

OT_SUCCESS if the AppInstallBlock was opened successfully and validated, an error code otherwise.

Comments

The reference returned by this function can be used as a parameter to any of the other functions that take an AIBFileRef.

OTError AIMscCloseAppInstallBlock(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

Return Values

OT_SUCCESS if the close succeeded, an error code otherwise.

Comments

None.

void AIMscGetAIBAppId(AIBFileRef aibFile, OTUInt8 pAIBAppId[16])

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AppId field in the AppInstallBlock.

Return Values

OT_SUCCESS if the value was successfully retrieved, an error code otherwise.

Comments

None.

void AIMscGetAIBVersionNo(AIBFileRef aibFile, OTUInt32 *pAIBVersionNo)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersionNo

[out] Returns the value of the VersionNo field in the AppInstallBlock.

Return Values

OT_SUCCESS if the value was successfully retrieved, an error code otherwise.

Comments

None.

void
AIMscGetAIBShouldReboot(
AIBFileRef aibFile,
OTBool *pAIBShouldReboot)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBShouldReboot

[out] Returns the value of the ShouldReboot flag in the AppInstallBlock.

Return Values

SUCCESS if the value was successfully retrieved, an error code otherwise.

Comments

None.

OTError
AIMscGetAIBAppName(
AIBFileRef aibFile,
char *pAIBAppName,
OTUInt16 *pSizeAIBAppName)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBAppName

[out] The value of the ApplicationName field in the AppInstallBlock is copied into the memory pointed to by this address (it will be null terminated).

pSizeAIBAppName

[in, out] On input, should point to the size of the memory at *pAIBAppName*. On output, will point to the total bytes needed to hold the entire string if OTERROR_BUFFER_TOO_SMALL is returned, otherwise is undefined.

Return Values

OT_SUCCESS if the value was successfully retrieved, OTERROR_BUFFER_TOO_SMALL if the buffer is too small to hold the entire string, or another error code otherwise.

Comments

None.

OTError
AIMscCheckAIBCompatibleOS(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

Return Values

OT_SUCCESS if the OS version was successfully retrieved and if the currently running OS is compatible.

Comments

This function will check if the currently installed operating system and service is compatible with the specified AppInstallBlock (using the compatibility information contained in the AppInstallBlock). If not, it will return OTERROR_INCOMPATIBLE_OS.

```
OTError
AIMscInstallAppFiles(
   AIBFileRef      aibFile,
   HKEY            spoofKey,
   const char      *installLogFile,
   OTBool          *pIsRebootNeeded)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

spoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file copying, FALSE otherwise.

Return Values

OT_SUCCESS if all file install operations succeeded, an error code otherwise.

Comments

This function will perform the file copies and add the file spoofing entries specified in the File section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

This function will not undo file copies or spoof entry additions if it fails. **AIMscUninstallAppFiles** should be called to do so.

```
OTError
AIMscUninstallAppFiles(
  AIBFileRef      aibFile,
  HKEY            spoofKey,
  const char      *installLogFile,
  OTBool          *pIsRebootNeeded)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

spoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file deletions, FALSE otherwise.

Return Values

OT_SUCCESS if enough of the file install operations were reversed so that re-installation will succeed and so that the system is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the file additions and remove the file spoof database entries specified in the install log file.

OTError
AIMscInstallAppVariables(
AIBFileRef aibFile,
const char *installLogFile,
const char *varRefCountFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

varRefCountFile

[in] A null-terminated string representing the path to the variable refcounting file for this user.

Return Values

OT_SUCCESS if all variable modifications succeeded, an error code otherwise.

Comments

This function will perform the add/remove variable (i.e. registry entry) changes specified in the Variable section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

This function will not undo registry modifications if it fails. **AIMscUninstallAppVariables** should be called to do so.

OTError
AIMscUninstallAppVariables(
AIBFileRef aibFile,
const char *installLogFile,
const char *varRefCountFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

varRefCountFile

[in] A null-terminated string representing the path to the variable refcounting file for this user.

Return Values

OT_SUCCESS if enough of the variable changes were reversed so that re-installation will succeed and so that the registry is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the add/remove variable (i.e. registry entry) changes specified in the install log file.

*OTError AIMscInstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)*

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

prefetchFile

[in] A null-terminated string representing the path to the prefetch file to be created.

Return Values

OT_SUCCESS if prefetch block installation succeeded, an error code otherwise.

Comments

This function will install the prefetch information contained in the Prefetch section of the AppInstallBlock into *prefetchFile.*

*OTError AIMscUninstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)*

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

pPrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be uninstalled.

Return Values

OT_SUCCESS if prefetch block uninstallation succeeded, an error code otherwise.

Comments

This function will remove the prefetch information stored at *prefetchFile.*

OTError AIMscCallCustomInstall(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Install()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Install()*.

OTError AIMscCallCustomUninstall(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Uninstall()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Uninstall()*.

OTError AIMscEnforceLicenseAgreement(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

OT_SUCCESS if the license agreement was successfully displayed and agreed to by the user, OTERROR_USER_CANCELLED, if it was displayed and not agreed to by the user, or an error code if it could not be displayed.

Comments

This function will extract the license agreement text included in the LicenseAgreement section of the AppInstallBlock and display it to the user. The user will

be given the option to agree or not agree to the license (probably via a pair of buttons in a dialog). OTERROR_USER_CANCELLED is returned if the users decides not to accept the license agreement.

OTError AIMscDisplayComment(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

OT_SUCCESS if the comment was successfully displayed, an error code otherwise.

Comments

This function will display to the user the comment included in the Comment section of the AppInstallBlock.

Component design

AIMsc does not have hard-coded knowledge regarding any of the standard registry and file locations used by AIM, which is why the functions in its interface take as inputs specifiers for filenames and base registry locations. Conversely, AIM itself has no knowledge of the internal structure of the AppInstallBlock file, which is why it must call AIMsc functions to work with such files.

Expansion is perform on registry entries and file paths containing certain variables, when they are read from the AppInstallBlock. These variables are defined in the Builder-LLD and will be recognized and expanded by AIM. (This includes file-spoof entries.)

AIM stores its data in the expected places for an eStream client component. All of the data AIM stores is user-specific, so it makes no use of the global locations defined for eStreams.

Registry keys

AIM stores its registry keys and values under:

HKEY_CURRENT_USER\SOFTWARE\Omnishift\eStream\AIM

This key will have its permissions modified so that ordinary users cannot modify the key (but the eStream client service will be given privileges so that it can do so). Here are the subkeys AIM places under this key:

"SpoofEntries"

Spoof entries are placed here. All spoofing is done globally, so there is no need to place it under an eStream-app specific key. Each value under this key is a pair of pathnames as follows:

<old-pathname> (REG_MULTI_SZ)

- <array of spoofed-pathnames>

The array contains one pathname for each installed eStream app that is spoofing the file. The file spoofer currently only spoofs to one of these files at a time, but this corresponds to a kind of refcounting of the spoof entry.

"<AppId>"

Every installed eStream app has its own subkey whose name is a string representation of its AppId, like so: "{00000000-0000-0000-0000-00000000000}". The values stored under each such key are:

AppId (REG_BINARY)
- AppId in binary form (16 bytes)

AppName (REG_SZ)
- name of the application (same as in the AppInstallBlock)

AppInstallBlockPath (REG_SZ)
- path to the AppInstallBlock for the application

AppInstallState (REG_DWORD)
- a value of 0 means app is installed, 1 means install is in progress, 2 mean uninstall is in progress.

Files

AIM stores per-user files at the following path:

(Path to the user's home directory)\Application Data\Omnishift\eStream\AIM

Here is what is stored in this directory:

RegistryRefCounts.dat - registry refcounts for eStream apps

<AppId GUID string> - a folder for each installed app

RegistryRefCounts.dat is a file that associates a refcount with every registry key that is added and every value that is modified during an eStream app install. Each null-terminated line in the file is of the form:

<hash of registry key\value path><tab><refcount of registry key\value>

For each installed application, a separate data folder is created. The name of the folder is the AppId of the application in GUID ASCII format, like so: "{00000000-0000-0000-0000-00000000000}". The files stored under each such folder are:

<GUID string>-AIB.dat - the AppInstallBlock file for the application
<GUID string>-Prefetch.dat - the prefetch data file for the application
InstallLog.txt - a generated log of what to do during uninstall

The Prefetch data file is simply an array of PrefetchItem structures (as described in the Data Structures section).

The InstallLog.txt is a list of undoable actions taken during installation. This log will be used during uninstall to determine which files and entries are safe to remove. Each line in the file contains one change, and is of the form:

ADDED or OVERWROTE or SPOOFED FILE <filename> (fully qualified)
ADDED or OVERWROTE KEY <keyname> (fully qualified)
ADDED or OVERWROTE VALUE <valuename> (fully qualified)

AIMInstallApplication Prototype

Installing an eStream application consists of the following steps:

1. Preparing for the installation
2. Displaying a license agreement to the user and having him agree to it
3. Installing all required local files and spoof entries for this app
4. Setting/removing registry entries as required
5. Initializing the prefetch data for this app
6. Performing any required custom installation tasks
7. Displaying the comment to the user if required
8. Completing the installation
9. Rebooting the computer if necessary

AIM's policy is that if it encounters any fatal error during the execution of AIMInstallApplication, it will attempt to undo everything it did before returning. AIM also gracefully handles aborted installs and uninstalls.

Step 1 – Preparing for the installation

First, AIM checks if the application is already installed by looking for an AppId registry key for the specified AppId. If found, then the AppInstallInProgress registry value is checked. If it exists and is 1, the user is asked if he wants to re-install, otherwise, he is asked to restart an aborted or damaged installation. If the user says no, **AIMInstallApplication** cleans up and exits with OTERROR_USER_CANCELLED.

Next, a free disk space check is performed to ensure that enough disk space is available for the install. The available free space must be at least twice the size of the AppInstallBlock to proceed. If not, **AIMInstallApplication** exits with OTERROR_VOLUME_FULL.

Next, an AppId folder is created for the app (described earlier), and the AppInstallBlock file is copied to this folder. Then AppId registry key is created and the four defined values created and initialized. The AppInstallState value in particular is set to 1 to indicate an install is in progress. Then, AIM then opens the AppInstallBlock using **AIMscOpenAppInstallBlock**, and calls **AIMscCheckAIBCompatibleOS** to check for OS compatibility. If any of these operations fail, **AIMInstallApplication** cleans up and exits with an error.

Step 2 – Displaying the license

AIMscEnforceLicenseAgreement is called to display the license text to the user and ask for his agreement. If the functions fails or if the user rejects the agreement, **AIMInstallApplication** cleans up and exits with OTERROR_USER_CANCELLED.

Step 3 – Installing local files

The install log file to be used for this application is created or open and truncated. **AIMscInstallAppFiles** is called to copy the install files to the computer and to create the spoof entries specified in the AppInstallBlock If the function fails, **AIMInstallApplication** cleans up and exits with an error.

If it succeeds, a boolean is returned indicating whether a reboot needs to occur due to shared files being overwritten. This value is remembered for use in step 9.

Step 4 – Modifying the registry

AIMscInstallAppVariables is called to perform the registry modifications specified in the AppInstallBlock. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 5 – Initializing prefetch data

AIMscInstallAppPrefetchFile is called to create and initialize the prefetch file for this application. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 6 – Performing custom install tasks

AIMscCallCustomInstall is called to extract the custom code .dll contained in the AppInstallBlock and to call the *Install()* function it exports. If **AIMscCallCustomInstall** fails, **AIMInstallApplication** cleans up and exits with an error.

Step 7 – Displaying a comment

AIMscDisplayComment is called to display any comment to the user contained in the appropriate section of the AppInstallblock. If this function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 8 – Completing the installation

The AppInstallInProgress registry value is set to 0 to indicate the install is complete. **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and any handles to open registry keys are also closed.

Step 9 – Rebooting the computer (if necessary)

If **AIMscInstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, no reboot is performed and the application is ready to be run. **AIMInstallApplication** exits returning OT_SUCCESS.

AIMUninstallApplication Prototype

Uninstalling an eStream application consists of the following steps:

1. Preparing for the uninstallation
2. Undoing all additions made to the registry during install
3. Removing all files copied during install and removing spoof entries for this app
4. Deleting the prefetch data for this application
5. Performing any required custom uninstallation tasks
6. Completing the uninstallation
7. Rebooting the computer if necessary

If the uninstallation fails for any reason, the called should tell the user that the uninstall has failed and that he should attempt to re-install the application before trying to uninstall again.

Step 1 – Preparing for the uninstallation

First, AIM checks if the application is already installed by looking for the AppId registry key corresponding to the specified AppId. If not found, then **AIMUninstallApplication** exits with OTERROR_ITEM_NOT_FOUND.

Then, the AppInstallState value is set to 2 to indicate an uninstall is in progress. **AIMscOpenAppInstallBlock** is called to open the AppInstallBlock at the path specified by the AppInstallBlockPath key. If this fails, then **AIMUninstallApplication** exits with an error.

Step 2 – Undoing registry modifications

AIMscUninstallAppVariables is called to reverse the registry additions made during installation. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 3 – Undoing file copies and removing spoof entries

AIMscUninstallAppFiles is called to delete the files copied during install and to remove the spoof entries written then. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

If it succeeds, a boolean is returned indicating whether a reboot needs to occur due to shared files being overwritten. This value is remembered for use in step 7.

Step 4 – Deleting profile/prefetch data

AIMscUninstallAppPrefetchFile will be called to remove the prefetch data stored for this application. Any failure is ignored.

Step 5 – Performing custom uninstall tasks

AIMscCallCustomUninstall is called to extract the custom code .dll contained in the AppInstallBlock and call the *Uninstall()* function it exports. If **AIMscCallCustomUninstall** fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 6 – Completing the uninstallation

AIMscGetAIBShouldReboot is called and the return value saved. Then **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and the AppId folder and all its contents are deleted. The AppId registry key and all its subkeys are deleted also. Any handles to open registry keys are closed. Any failures here are ignored.

Step 7 – Rebooting the computer (if necessary)

If **AIMscUninstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, the uninstallation is complete. **AIMUninstallApplication** exits returning SUCCESS.

AIMsc Function Prototypes

Prototypes for the AIMsc functions declared earlier are given in this section.

OTError
AIMscOpenAppInstallBlock(const char *pathToAIB, AIBFileRef *pAIBFile)

A class AppInstallBlock object is created, and the appropriate member function is called to open and verify the AppInstallBlock. An opaque pointer to this object is returned in the *pAIBFile* parameter.

OTError
AIMscCloseAppInstallBlock(AIBFileRef aibFile)

The AppInstallBlock class object pointed to by *aibFile* is deleted.

void
AIMscGetAIBAppId(AIBFileRef aibFile, OTUInt8 pAIBAppId[16])

void
AIMscGetAIBVersionNo(AIBFileRef aibFile, OTUInt32 *pAIBVersionNo)

void
AIMscGetAIBShouldReboot(
AIBFileRef aibFile,
OTBool *pAIBShouldReboot)

OTError
AIMscGetAIBAppName(
AIBFileRef aibFile,
const char *pAIBAppName,
OTUInt16 *pSizeAIBAppName)

These four functions are trivial. They directly map to method calls on the AppInstall-Block class object pointed to by *aibFile*, which retrieves the associated header field of the AppInstallBlock. (See the interface declaration for **AIMscGetAIBAppName** for details on its calling logic.)

OTError
AIMscCheckAIBCompatibleOS(AIBFileRef aibFile)

This function will call an API such as GetVersionEx (for Windows) to determine the currently running operating system. The OS version is then compared to the OS version stored in the AppInstallBlock by calling a method on the AppInstallBlock object pointed to by *aibFile*.

OTError
AIMscInstallAppFiles(

```
    AIBFileRef      aibFile,
    HKEY            spoofKey,
    const char      *installLogFile,
    OTBool          *pIsRebootNeeded)
```

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the retrieve the entries in the File section of the AppInstallBlock.

The File section is organized as a series of trees, with directories as non-leaf nodes and plain files as leaf nodes. All nodes are stored contiguously according to the pre-order traversal of the trees. The AIMsc code does not parse the section itself, but relies upon the code in the provided AppInstallBlock class to do the unpacking for it.

For every file copied or spoof entry added by the algorithm, an entry is made to the file at *InstallLogFile*.

The algorithm is as follows:

```
while there are file entries to read in the File section
    read an entry

    if entry filename contains Builder/AIM defined variables
        replace variables with local expansions

    if the file node is a spoof entry
        call HandleSpoofEntry()
    else
        call HandleCopyEntry()
```

Here is HandleSpoofEntry():

```
if filename already exists
    if existing version is earlier or version can't be read
```

```
        mark for spoofing
    else  // filename does not exist
        create zero-length file at filename
        mark for spoofing

    if marked for spoofing
        create/modify spoof entry under SpoofKey
```

Here is HandleCopyEntry():

```
if filename already exists
    increment shared file ref count in registry
    if existing version is earlier or version can't be read
        mark for copy
else  // filename does not exist
    mark for copy

if marked for copy
    attempt to copy node file to client computer
    if copy fails
        tell system to perform copy at reboot
```

The *pIsRebootNeeded* argument will be set to TRUE if any file copies were scheduled to happen at reboot, FALSE otherwise. Additionally, if any spoof entries were added, then an IOCTL will be sent to the spoof driver asking it to reload the spoof database.

The shared file reference count mentioned in the algorithm above is stored in a standard place in the Windows registry. AIM will create or increment this reference count for every non-spoofed file included in the AppInstallBlock (they can all be potentially shared since they will be placed outside of the eStream app directory). Each such file has an associated REG_DWORD value under the key at:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
SharedDLLs

Even though the subkey name is called 'SharedDLLs', this key is used by all well-behaved applications to refcount all shared files. The value's name is the path to the file and the value's data is a integer that is the reference count for this file.

OTError
AIMscUninstallAppFiles(
AIBFileRef aibFile,

HKEY spoofKey,
const char *installLogFile,
OTBool *pIsRebootNeeded)

Currently, the *aibFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppFiles** is simple. It iterates over the change entries contained in the log file, and undoes file copies and spoof entry additions when it is safe to do so. Here is the algorithm:

```
while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED or OVERWROTE <filename>"
        decrease refcount of file
        if refcount is 0
            attempt to delete file
            if deletion fails
                tell system to perform deletion at reboot

    else if the entry is of the form "SPOOFED <filename>"
        decrease refcount of spoof entry for this filename
        if refcount is 0
            delete/modify the spoof entry
            if 0-byte placeholder at filename still exists
                delete it
```

A failure to delete a file or to schedule its deletion will not cause **AIMscUninstallAppFiles** to fail.

OTError
AIMscInstallAppVariables(
AIBFileRef aibFile,
const char *installLogFile,
const char *varRefCountFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the retrieve the entries in the Variables section of the AppInstallBlock.

The Variable section is organized as a series of trees, similar to how the File section is organized. There are two types of nodes, key nodes and value nodes. Registry keys can contain other keys and registry values, while registry values are just name/data pairs that are stored in keys. The AIMsc code does not parse the section itself, but relies upon the code in the provided AppInstallBlock class to do the unpacking for it.

Each entry can be either an add key/value entry, or a delete key/value entry. For every change made, an entry is made to the file at *installLogFile*. Also, registry key and value additions made by eStream are refcounted in the file at *varRefCountFile*, but deletions are not refcounted. (Only deletions during uninstall are refcounted).

The algorithm is as follows:

```
while there are variable entries to read in the Variable section
    read an entry

    if entry name contains Builder/AIM defined variables
        replace variables with local expansions

    if the variable node is a key entry
        call HandleKeyEntry()
    else
        call HandleValueEntry()
```

Here is HandleKeyEntry():

```
if the key name is not fully qualified
    error
if this is an add key entry
    if key doesn't already exist
        create key
else    // it's a delete key entry
    delete the key
increment key refcount
```

Here is HandleCopyEntry():

```
if this is an add value entry
    add the value and its data
    increment value refcount
else    // it's a delete value entry
```

delete the value

UINT32
OTError
AIMscUninstallAppVariables(
AIBFileRef aibFile,
const char *installLogFile,
const char *varRefCountFile)

Currently, the *aibFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppVariables** iterates over the change entries contained in the log file, and undoes registry key and value additions when it is safe to do so. The biggest complication is that it must handle COM entries specially, which requires two passes. Here is the algorithm:

```
Pass 1:

while there are change entries in the log file
    read the next entry

if this is an entry of the form "ADDED/OVERWROTE KEY <CLSID
keyname>"
    check the refcount of the executable that provides this CLSID
    if the refcount is > 0
        mark this CLSID as sacred

Pass 2:

while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED/OVERWROTE KEY <keyname>"
        decrement keyname refcount
        if keyname refcount is 0
            if keyname is not a COM key with a sacred CLSID value
                delete it and all its subkeys and values
```

else if the entry is of the form "ADDED/OVERWROTE VALUE <value-name>"

decrement valuename refcount

if valuename refcount is 0

if value is not part of a COM key with a sacred CLSID value

delete it

A failure to delete one or more registry entries will not cause **AIMscUninstallAppFiles** to fail.

OTError
AIMscInstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the data in the Prefetch section of the AppInstallBlock. The data are written out into the file at *prefetchFile* as an array of PrefetchItem structures. Any existing file at *PrefetchFile* is overwritten.

Next, the Prefetch component is called to set up an association between the new application and its prefetch file.

OTError
AIMscUninstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)

The file at *PrefetchFile* is deleted. (No call need be made to the Prefetch component.)

OTError
AIMscCallCustomInstall(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the data in the Code section of the AppInstallBlock. This data is written out as a .dll library. This library is loaded and the *Install()* function export is called (and its return value returned).

OTError
AIMscCallCustomUninstall(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the data in the Code section of the AppInstallBlock. This data is written out as a .dll library. This library is loaded and the *Uninstall()* function export is called (and its return value returned).

OTError
AIMscEnforceLicenseAgreement(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the text in the License Agreement section of the AppInstallBlock. The text is displayed to the user in a dialog box with two buttons, 'Agree' and 'Disagree'.

OTError
AIMscDisplayComment(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the text in the Comment section of the AppInstallBlock. The text is displayed to the user in a dialog box.

Testing design

Unit testing plans

AIM will be tested by a program that generates AppInstallBlocks designed to stress the component. AIM will be asked to install the given AIB and if successful, the resulting state of the system will be compared to the expected state had all the files and variables been installed correctly. An uninstall will then be performed and the system state also checked.

The focus of the testing will obviously be on the File and Variable sections. The other sections such as Code and Comments will be stressed also, but their boundary conditions are much simpler.

AIM's ability to gracefully handle aborted installs and uninstalls will also be tested.

Stress testing plans

The program described above can be deliberately tuned to create AppInstallBlocks of unusual size and organization. For example, AppInstallBlocks with thousands of files and registry entries, or files and entries with unusually long names, etc.

Coverage testing plans

In addition to the stress testing, deliberately malformed AppInstallBlocks will be generated by the test program to hit as much error-handling code as possible. AIM's data files and registry entries can also be deliberately mangled to help achieve this effect.

Cross-component testing plans

As soon as they are available, Builder-generated AppInstallBlocks will be tested to verify that the AIM is compatible with the Builder's output. As soon as the LSM and a browser plugin are available, the communication path from browser to LSM to AIM will be

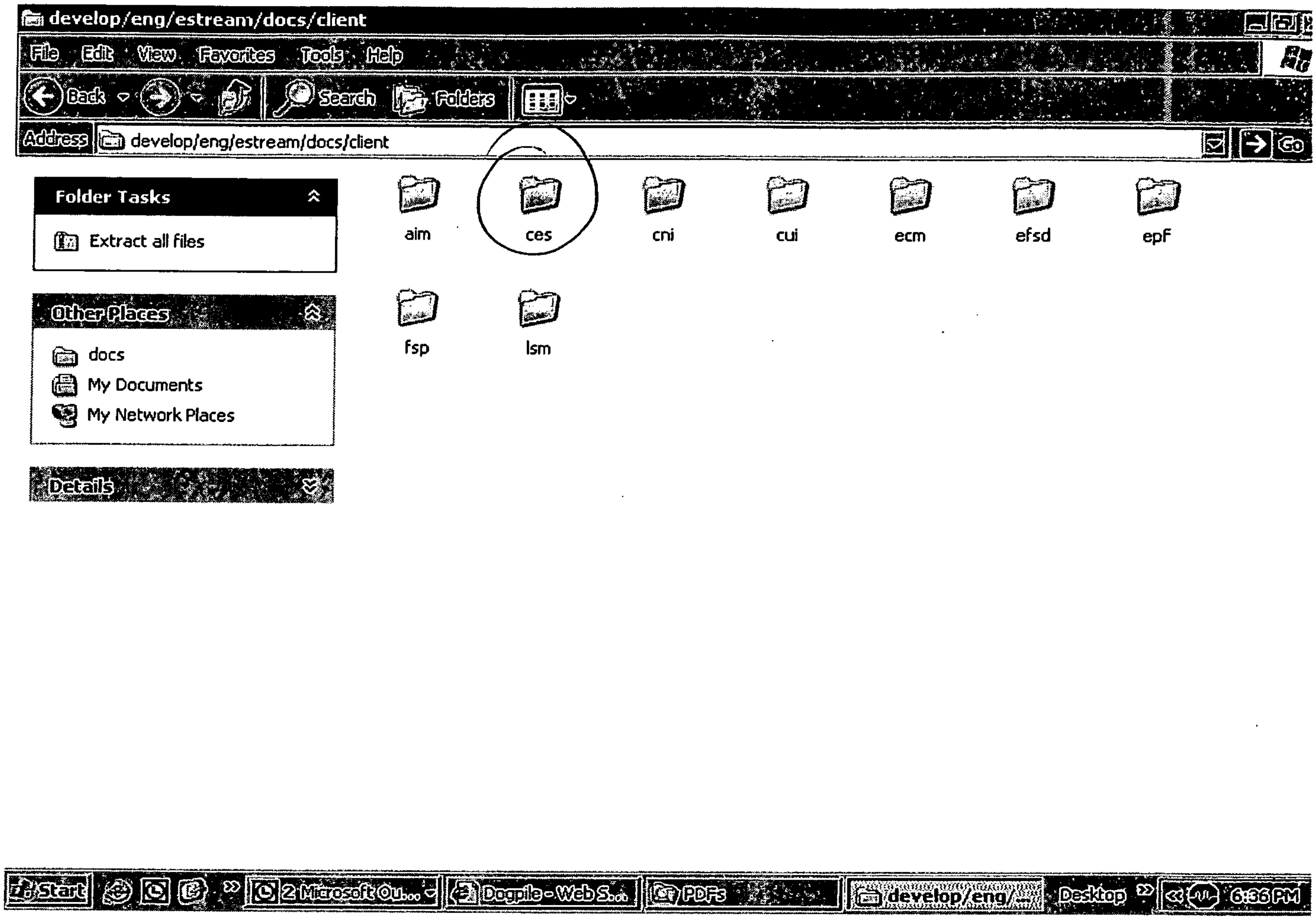
tested. As soon as the file spoofer is available, compatibility with the file spoof entries that AIM makes will be tested.

Upgrading/Supportability/Deployment design

AIM will make use of the eStream logging facility to record information about errors and other unusual conditions that occur. The log file will be useful for diagnosing problems that occur during testing and in real world situations.

If the AIM component is upgraded, it must still be able to uninstall any eStream applications installed at the time of upgrade. This entails being able to interpret old AIM registry entries and data files, including the AppInstallBlock. This is more a concern for future designers of the AIM component, however.

Open Issues


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back
Search
Folders
Address
develop/eng/estream/docs/client
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epF
fsp
lsm
Start
2 Microsoft Ou...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
6:36 PM

[/STANDARD.01]

eStream Client Installer/Uninstaller & Startup/Shutdown Low Level Design

*Anne Holler * [redacted] * Version 2.9*

Functionality

The Client Installer/Uninstaller [CIN] software is used to setup/remove a client's capability to eStream applications via installing/uninstalling eStream client software. [eStream client software patches are handled via special encapsulated executables distributed with patches & are not discussed herein.] eStream client software consists of:

- kernel-mode drivers [EFSD, FileSpoofer, NoCluster]
- user-mode modules comprising the privileged eStream Client Executable [ECE]:
    - Client EStream Startup/Shutdown [CES]
    - Estream Prefetch Fetch [EPF]
    - License Subscription Manager [LSM]
    - Application Install Manager [AIM]
    - EStream Cache Manager [ECM]
    - Client Network Interface [CNI]
    - Client User Interface [CUI]
- user-mode non-privileged executable eSUser.exe, which is run by an eStream user at log in, at log out, & optionally at other times
- user-mode non-privileged Client Browser Module [CBM] called eSCBM.exe, which fields notification messages from ASP servers [Design reviewers suggested investigating combining eSUser.exe & eSCBM.exe, migrating CUI into the module, & allowing MIME messages to control UI operations]

The Component Design section of the document first addresses CIN.

The Client EStream Startup/Shutdown [CES] software comprises ECE's main; it calls client components' initialization routines, it orchestrates activating various eStream client drivers, and it handles shutting the eStream client software down. The Component Design section of the document next addresses CES.

To understand the proposed operation of the CIN and CES, the reader is expected to be familiar with all client LLD design material.

Data Definitions

CIN is a standalone process and does not share any data definitions with other portions of eStream; registry keys used for communicating data are defined in the Component Design section. CES obtains, verifies, tracks, and communicates information about the

eStream user (his windows account name and a handle for his display) for passing to LSM and CUI.

Interface Definitions

Registry keys and IOCTL interfaces are described in the Component Design section. Below are the procedural interfaces.

From CES:

- CNIInitialize(void)
- EPFInitialize(void)
- ECMInitialize(void)
- CUIInitialize(IN HandleToDesktop)
- LSMInitialize(IN WindowsUserName)
- LSMUpdateAllSubscriptionStatus(void)

To CES:

- CESSetMode(IN ModeRequested) -- Modes are STANDBY, READY, ACTIVE

Component Design

Client Installer/Uninstaller [CIN]

This section outlines the operations involved in installation & uninstallation of the four client packages listed above, after describing special attributes of the ECE and CBM that are relevant to CIN. [BTW, to put our installation into perspective, SoftwareWow installs 19 files: 4 EXEs, 1 DLL, 6 SYSs, 2 VXDs, 4 WAVs, and 2 HLP/CNTs.]

Attributes of eStream Client Executable

The ECE is chiefly a performance-critical extension of the eStream client file system and certain parts of ECE have been identified as potentially moving to kernel-mode for performance reasons, including ECM, EPF, LSM and possibly CNI. The ECE contains certain administrative modules for which it is desirable to execute in the context of a separate privileged account, including AIM, LSM, and possibly CES. Several client tasks [eSUser.exe and eSCBM.exe] need to communicate with ECE asynchronously. Given this set of characteristics. ECE will be implemented as a daemon process launched at boot time.

On W2K/WNT, ECE is planned to be a windows service program [ref: MSDN Library \Platform SDK\Windows Base Services\Executables\Services]. ECE starts running when the system boots and it runs under a privileged account to allow it to hide various data for privacy and piracy protection. ECM is not a COM server; it does not contain any inter-

faces intended to be exported for running as instances inside a COM client. It is also not a COM client; it does not manage compound documents or import functionality from any COM servers. ECE contains two message queues into which client tasks can post messages. One message queue [CBMmessages] handles messages from eSCBM.exe & the other message queue [USRmessages] handles messages from eSUser.exe. [Guidance on selecting between Clipboard, COM, DDE, FileMapping, Mailslot, Pipes, RPC, Sockets, & WM_COPYDATA for IPC implementation is found in MSDN Library \Platform SDK \Windows Base Services \Interprocess Communication\Choosing an IPC Mechanism.]

Attributes of Client Browser Module

The CBM is a standard browser helper application, i.e., it is a Windows32 application that is registered to handle a particular MIME type embedded in HTML. Helper apps are supported in both Internet Explorer 4+ and Netscape Navigator 4+. To register a helper application for handling a certain MIME type, one sets up the following:

- key for mime extension under HKLM\MIME\Database\Content Type
- key under HKLM\SOFTWARE\Classes for class type which application handles
- key under HKCR for file extension which application handles

[BTW, SoftwareWow creates these entries to register a helper app to handle .wow files.]

CBM talks with ECE by placing messages in the CBMmessage message queue.

eStream Client Installation

A standard installation tool [likely InstallShield] will be used to create the installation package. Installation images for distribution on CD-ROM and via the internet will be provided. Uninstallation via the "Add/Remove Programs" will be supported.

BTW, the windows installer is discussed in MSDNLibrary/ Platform SDK/ Management Services/ Setup/Windows Installer. The windows installer is a privileged service. A "managed application" is an application that is installed on a per-machine (not per-user) basis & which requires elevated privileges for installation. The application installation is marked via the ALLUSERS property as to its installation privilege needs. If app is per-machine, but user is unprivileged, installation will fail unless the AlwaysInstallElevated registry key is set.

Installation will be performed under administrative privileges. To handle the situation of residual partial installation due to a failed/aborted previous installation, there is always an [invisible] uninstall-like step that cleans up before an install. eStream client software (and supported eStream applications) expect a client's DLLs to be at a certain revision level; the installation process will check them & if they are not at that level or higher, the installation will include upgrading them. EStream-specific installation activities are described in the paragraphs that follow.

The user is asked where he would like to install eStream-related files, with the default location set to C:\Program Files\Omnishift\eStream. The installation path chosen is written to the registry key HKLM\Software\Omnishift\eStream\InstallPath. A directory for TEMP files is created at InstallPath\temp and the temp path is written to the registry key HKLM\Software\Omnishift\eStream\TempPath. The estream version number is written to HKLM\Software\Omnishift\eStream\Version. The user is asked the drive letter/UNC path at which to mount the eStream file system & this is put into the registry at HKLM\Software\Omnishift\eStream\FileMountLocation.

The kernel-mode drivers (NoCluster, EFSD, FileSpoofer) are installed in the system directory & set up via registry entries for automatic loading at boot time.

- Copy {NoCluster.sys,efsd.sys,fspoof.sys} to %SystemRoot%\system32\drivers
- Set up registry to specify loading at boot time
    - Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_DRIVER (as appropriate)
        - Set START to SERVICE_SYSTEM_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL

Please note that the names NoCluster.sys & fspoof.sys are used above for the reader's convenience. For the product, we may obscure their purpose via naming them something like OTI001.sys & OTI002.sys.

The ECE is installed as a windows service & set up for automatic loading at boot time.

- Create a special privileged account under which the ECE service will run
- Copy estream.exe to %SystemRoot%\system32\estream.exe
- Set up registry to specify load at boot time, specify to run under privileged accnt
    - Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_WIN32_OWN_PROCESS
        - Set START to SERVICE_AUTO_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL
        - Set IMAGEPATH to %SystemRoot%\system32\estream.exe

eSUser.exe is copied to location in HKLM\Software\Omnishift\eStream\InstallPath.
eSCBM.exe is copied to location in HKLM\Software\Omnishift\eStream\InstallPath.
eStream user-specific actions are set up:

- During installation, user is asked if he/she would like eStream to be automatically activated at log in to windows (preferred). The answer determines which value (ready|activate) is sent to eSUser.exe when it is run at the user's login.

- User account is set up to run "eSUser.exe ready|activate" whenever user logs in to windows.
- User gets a Start\Program link which runs "eSUser.exe activate". (BTW, both SoftwareWow & NewMoon have Start\Program entries.)
- User account is set up to run "eSUser.exe standby" whenever user logs out of windows.
- Registry keys are set up to hold user's ASP Data Set and Subscription Data Set.

eStream Client Uninstallation

eStream-specific installation activities are undone at uninstall; the files NoCluster.sys, efsd.sys, fspoof.sys, estream.exe, eSUser.exe, & eSCBM.exe are deleted, the relevant registry keys are removed, and the user-specific operations for each eStream user on this client are yanked. We need to decide if apps should be [automatically] uninstalled when user requests that eStream client software be uninstalled.

Client eStream Startup/Shutdown [CES]

The actions of CES need to be understood in the context of the overall operation of the eStream client software; hence, the description of CES below is embedded in a general discussion of the operation of ECE. This is followed by a section considering the issues surrounding usability when eStream client software is in READY (i.e., not yet active) mode for a user.

Operation of ECE

At system boot time, the kernel-mode drivers are loaded, but they are not performing eStream-related activities. ECE is loaded as a service program belonging to the eStream privileged account set up at install time and is ready to receive messages in its message queues. The eStream client software is considered to be in STANDBY mode. No systray icon is displayed.

When an eStream user logs in to the client's console, "eSUser.exe ready|activate" is run and it sends a READY message to ECE on behalf of this user. [Please note that a client system only supports one active eStream user at a time.] When ECE receives the message, it:

- runs initialization routines for each client module that has them [expected order from lowest to highest level: CNIInitialize, LSMInitialize, EPFInitialize, ECMInitialize, CUIInitialize]
- calls the LSM interface that reads in this user's ASP & Subscription data [stored in privileged registry entries] and that performs a synchronize operation
- brings up a systray icon on the eStream user's desktop

The eStream client is now considered to be in READY mode on behalf of this user; it remains in READY mode if eSUser.exe was invoked with the ready parameter.

Whenever eSUser.exe is run with the activate parameter (either automatically at the user's windows login or manually via a Start/Programs entry selection), it sends an ACTIVATE message to ECE on behalf of the user. [A discussion of issues related to eStream being activated automatically in other situations is included below.] The eStream client software moves from READY mode to ACTIVE mode for the user in question, performing the following actions:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to begin eStream-related activities:
    - NoCluster: CES sends IOCTL to activate [or may be active at boot]
    - EFSD: CES sends IOCTL_EFS_START_FS
    - FileSpoofer: CES sends IOCTL_FS_START_SPOOFING
- If any ECE routines need to execute separate initialization on client activation, those routines are called at this point.

When eStream client software is in ACTIVE mode for a user, that user can seamlessly run eStream applications; when it is in READY mode for a user, eStream applications cannot be executed immediately. Issues involving eStream client software automatically transitioning from READY to ACTIVE when a user attempts to run an eStream application or providing a meaningful error message to the user if that automatic transition is not supported are considered in the next section.

ECE may be deactivated automatically at eStream user logout [eSUser.exe standby] or manually on demand via the CUI (mode request for READY). When the ECE is deactivated, the following events are performed:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to stop eStream-related activities:
    - NoCluster: CES sends IOCTL to deactivate [or may continue active]
    - EFSD: CES sends IOCTL_EFS_SHUTDOWN_FS
    - FileSpoofer: CES sends IOCTL_FS_STOP_SPOOFING
- If any ECE routines need to execute separate termination on client deactivation, those routines are called at this point.

ECE brings up a browser to provide web-based ABOUT/HELP information for eStream and to allow the user to interact with his ASP to obtain/modify account-specific data. For eStream 1.0, this communication is handled via the standard mechanism used by currently-available windows application that provides web access, i.e., the user's default browser is invoked in a separate window with a particular URL and set of parameters

[decided at 9/7 meeting of Lacky, Bhaven, Manoj, and me]. A common way to accomplish this is to execute a variant of the ShellExecute command.

Automatic Transition from READY to ACTIVE mode Plus Error Handling

If the eStream client software is not active when an eStream user tries to execute an eStream application, it is attractive to either give a meaningful error message or to automatically activate eStream. [To my knowledge, doing either is an eFSD issue; I am including a discussion of the issue here since the topic seems to be open at this time.] To give a meaningful error message, eFSD needs to see the relevant file reference, meaning that either the eStream drive letter needs to exist or eFSD needs to be registered as a UNC provider; otherwise, we may give a somewhat inscrutable error message. To automatically transition from READY to ACTIVE when we see a relevant file reference, we need to ensure that file spoofing is not needed prior to eFSD receiving the reference to the inactive eStream file system for apps of interest. Please note that it is not required that eStream client software be activated automatically if CBM finds it inactive.

Testing Design

Unit and Coverage testing plans

For CIN, an installation/deinstallation of stub versions of the various eStream client software will be developed. For CES, a test harness will be developed to test all CES operational modes (standby, transition to ready, transition to active, transition to inactive, shutdown) and is expected to achieve essentially 100% PFA coverage.

Stress testing plans

CIN stress testing will involve testing a myriad of software and hardware system configurations. CES stress testing will involve testing behavior in the presence of various problems with the eStream client software configuration and for various user situations.

Cross-component testing plans

CIN is expected to be incorporated with the first set of working eStream 1.0 software & installing/uninstalling should form a key step in the daily automated testing of eStream 1.0 going forward. CES is also expected to be included in the first set of working eStream 1.0 software and should also get a daily workout.

Upgrading/Supportability/Deployment Design

Both CIN and CES should be developed with an eye to minimizing issues related to upgrading/supportability/deployment. The bar is very high for these components to contain heavy error checking and reporting.

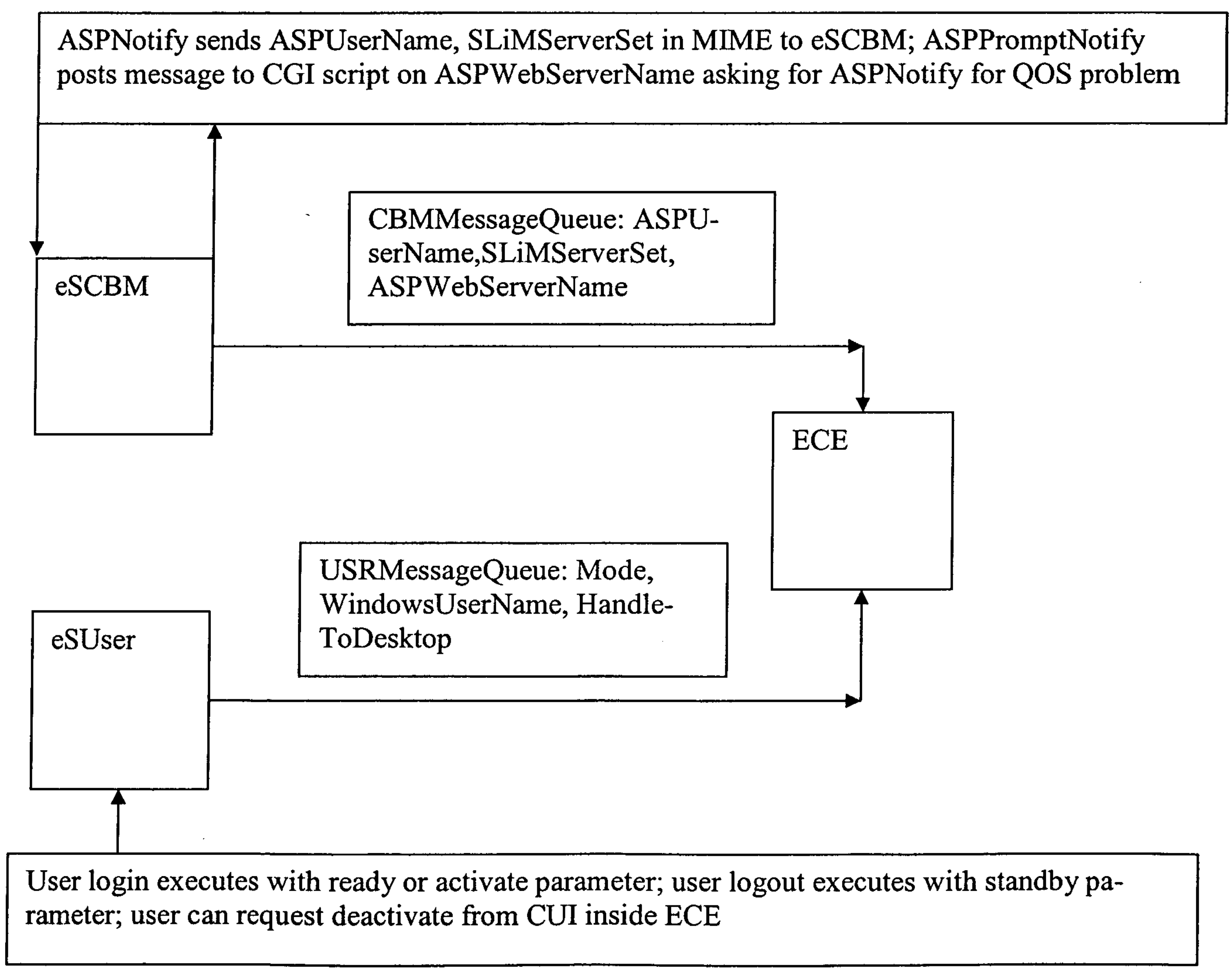
Attachment 1: eStream Client Mode Transitions

Current Mode	Mode Request	Normal Mode Transition
• Standby	Ready	Transition to Ready
• Standby	Active	Transition to Ready, then to Active
• Ready	Active	Transition to Active
• Ready	Standby	Transition to Standby
• Active	Ready	Transition to Ready
• Active	Standby	Transition to Ready, then Standby

Attachment 2: Running scripts at Logon & at Logout

Windows2000 supports executing scripts & programs at user logon & logoff. Registry keys control whether the executions are synchronous or asynchronous wrt logon or logoff & whether the processes interact with the user. To install these processes, one runs start/run/gpedit.msc & chooses UserConfiguration/WindowsSettings/Scripts(Logon/Logoff).

Attachment 3: eStream Client IPC



Client Installer/Uninstaller & Estream Startup/Shutdown Straw Man
Anne Holler *[redacted]* *Version 1.6*

Introduction

The Client Installer/Uninstaller [CIN] software is used to setup/remove a client's capability to eStream applications via installing/uninstalling eStream client software. [eStream client software patches are handled via special encapsulated executables distributed with patches & are not discussed herein.] eStream client software consists of:

- kernel-mode drivers [EFSD, FileSpoofer, NoCluster]
- user-mode modules comprising the privileged eStream Client Executable [ECE]:
    - Client EStream Startup/Shutdown [CES]
    - Estream Prefetch Fetch [EPF]
    - License Subscription Manager [LSM]
    - Application Install Manager [AIM]
    - EStream Cache Manager [ECM]
    - Client Network Interface [CNI]
    - Client User Interface [CUI]
- user-mode non-privileged executable eSLogin.exe, which is run by an eStream user at log in, at log out, & optionally at other times
- user-mode non-privileged Client Browser Module [CBM] called eSCBM.exe, which fields notification messages from ASP servers

This document first addresses CIN.

The Client EStream Startup/Shutdown [CES] software comprises ECE's main; it calls client components' initialization routines, it orchestrates activating various eStream client drivers, and it handles shutting the eStream client software down. This document next addresses CES.

To understand the proposed operation of the CIN and CES, the reader is expected to be familiar with all client LLD design material.

Client Installer/Uninstaller [CIN]

This section outlines the operations involved in installation & uninstallation of the four client packages listed above, after describing special attributes of the ECE and CBM that are relevant to CIN. [BTW, to put our installation into perspective, SoftwareWow installs 19 files: 4 EXEs, 1 DLL, 6 SYSs, 2 VXDs, 4 WAVs, and 2 HLP/CNTs.]

Attributes of eStream Client Executable

The ECE is chiefly a performance-critical extension of the eStream client file system and certain parts of ECE have been identified as potentially moving to kernel-mode for performance reasons, including ECM, EPF, LSM and possibly CNI. The ECE contains certain administrative modules for which it is desirable to execute in the context of a

separate privileged account, including AIM, LSM, and possibly CES. Several client tasks [eSLogin.exe and eSCBM.exe] need to communicate with ECE asynchronously. Given this set of characteristics. ECE will be implemented as a daemon process launched at boot time.

On W2K/WNT, ECE is planned to be a windows service program [ref: MSDN Library \Platform SDK\Windows Base Services\Executables\Services]. ECE starts running when the system boots and it runs under a privileged account to allow it to hide various data for privacy and piracy protection. ECM is not a COM server; it does not contain any interfaces intended to be exported for running as instances inside a COM client. It is also not a COM client; it does not manage compound documents or import functionality from any COM servers. ECE contains two message queues into which client tasks can post messages. One message queue [CBMmessages] handles messages from eSCBM.exe & the other message queue [LGNmessages] handles messages from eSLogin.exe. [Guidance on selecting between Clipboard, COM, DDE, FileMapping, Mailslot, Pipes, RPC, Sockets, & WM_COPYDATA for IPC implementation is found in MSDN Library \Platform SDK \Windows Base Services \Interprocess Communication\Choosing an IPC Mechanism.]

Attributes of Client Browser Module

The CBM is a standard browser helper application, i.e., it is a Windows32 application that is registered to handle a particular MIME type embedded in HTML. Helper apps are supported in both Internet Explorer 4+ and Netscape Navigator 4+. To register a helper application for handling a certain MIME type, one sets up the following:

- key for mime extension under HKLM\MIME\Database\Content Type
- key under HKLM\SOFTWARE\Classes for class type which application handles
- key under HKCR for file extension which application handles

[BTW, SoftwareWow creates these entries to register a helper app to handle .wow files.] CBM talks with ECE by placing messages in the CBMmessage message queue.

eStream Client Installation

A standard installation tool [likely InstallShield] will be used to create the installation package. Installation images for distribution on CD-ROM and via the internet will be provided. Uninstallation via the "Add/Remove Programs" will be supported.

Installation will be performed under administrative privileges. To handle the situation of residual partial installation due to a failed/aborted previous installation, there is always an [invisible] uninstall-like step that cleans up before an install. eStream client software (and supported eStream applications) expect a client's DLLs to be at a certain revision level; the installation process will check them & if they are not at that level or higher, the installation will include upgrading them. eStream-specific installation activities are described in the paragraphs that follow.

The user is asked where he would like to install eStream-related files, with the default location set to C:\Program Files\Omnishift\eStream1.0. The installation path chosen is written to the registry key HKLM\Software\Omnishift\eStream1.0\InstallPath. A directory for TEMP files is created at InstallPath\temp and the temp path is written to the registry key HKLM\Software\Omnishift\eStream1.0\TempPath. The estream version number is written to HKLM\Software\Omnishift\eStream1.0\eStreamVersion.

The kernel-mode drivers (NoCluster, EFSD, FileSpoofer) are installed in the system directory & set up via registry entries for automatic loading at boot time.

- Copy {NoCluster.sys,efsd.sys,fspoof.sys} to %SystemRoot%\system32\drivers
- Set up registry to specify loading at boot time
    - Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_DRIVER (as appropriate)
        - Set START to SERVICE_SYSTEM_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL

Please note that the names NoCluster.sys and fspoof.sys are used above as a convenience to the reader. For the product, we may obscure their functionality via naming them something like OTI001.sys and OTI002.sys.

The ECE is installed as a windows service & set up for automatic loading at boot time.

- Create a special privileged account under which the ECE service will run
- Copy estream.exe to %SystemRoot%\system32\estream.exe
- Set up registry to specify load at boot time, specify to run under privileged accnt
    - Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_WIN32_OWN_PROCESS
        - Set START to SERVICE_AUTO_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL
        - Set IMAGEPATH to %SystemRoot%\system32\estream.exe

eSLogin.exe is copied to location in HKLM\Software\Omnishift\eStream1.0\InstallPath.

eSCBM.exe is copied to location in HKLM\Software\Omnishift\eStream1.0\InstallPath.

eStream user-specific actions are set up:

- During installation, user is asked if he/she would like eStream to be automatically activated at log in to windows (preferred). The answer determines which value (ready|activate) is sent to eSLogin.exe when it is run at the user's login.
- User account is set up to run "eSLogin.exe ready|activate" whenever user logs in to windows.
- User gets a Start\Program link which runs "eSLogin.exe activate". (BTW, both SoftwareWow & NewMoon have Start\Program entries.)
- User account is set up to run "eSLogin.exe deactivate" whenever user logs out of windows.
- Registry keys are set up to hold user's ASP Data Set and Subscription Data Set.

eStream Client Uninstallation

eStream-specific installation activities are undone at uninstall; the files NoCluster.sys, efsd.sys, fspoof.sys, estream.exe, eSLogin.exe, & eSCBM.exe are deleted, the relevant registry keys are removed, and the user-specific operations for each eStream user on this client are yanked.

Client eStream Startup/Shutdown [CES]

The actions of CES need to be understood in the context of the overall operation of the eStream client software; hence, the description of CES below is embedded in a general discussion of the operation of ECE. This is followed by a section considering the issues surrounding usability when eStream client software is in READY (i.e., not yet active) mode for a user.

Operation of ECE

At system boot time, the kernel-mode drivers are loaded, but they are not performing eStream-related activities. ECE is loaded as a service program belonging to the eStream privileged account set up at install time and is ready to receive messages in its message queues. The eStream client software is considered to be in STANDBY mode. No systray icon is displayed.

When an eStream user logs in to the client's console, "eSLogin.exe ready|activate" is run and it sends a READY message to ECE on behalf of this user. [Please note that a client system only supports one active eStream user at a time.] When ECE receives the message, it:

- runs initialization routines for each client module that has them [expected order from lowest to highest level: CNIStartup, LSMStartup, EPFStartup, ECMStartup, CUIStartup]
- calls the LSM interface that reads in this user's ASP & Subscription data [stored in privileged registry entries] and that performs a synchronize operation
- brings up a systray icon on the eStream user's desktop

The eStream client is now considered to be in READY mode on behalf of this user; it remains in READY mode if eSLogin.exe was invoked with the ready parameter.

Whenever eSLogin.exe is run with the activate parameter (either automatically at the user's windows login or manually via a Start/Programs entry selection), it sends an ACTIVATE message to ECE on behalf of the user. [A discussion of issues related to eStream being activated automatically in other situations is included below.] The eStream client software moves from READY mode to ACTIVE mode for the user in question, performing the following actions:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to begin eStream-related activities:
  - NoCluster: CES sends IOCTL to activate [or may be active at boot]

    - EFSD: CES sends IOCTL_EFS_START_FS
    - FileSpoofer: CES sends IOCTL_FS_START_SPOOFING
- If any ECE routines need to execute separate initialization on client activation, those routines are called at this point.

When eStream client software is in ACTIVE mode for a user, that user can seamlessly run eStream applications; when it is in READY mode for a user, eStream applications cannot be executed immediately. Issues involving eStream client software automatically transitioning from READY to ACTIVE when a user attempts to run an eStream application or providing a meaningful error message to the user if that automatic transition is not supported are considered in the next section.

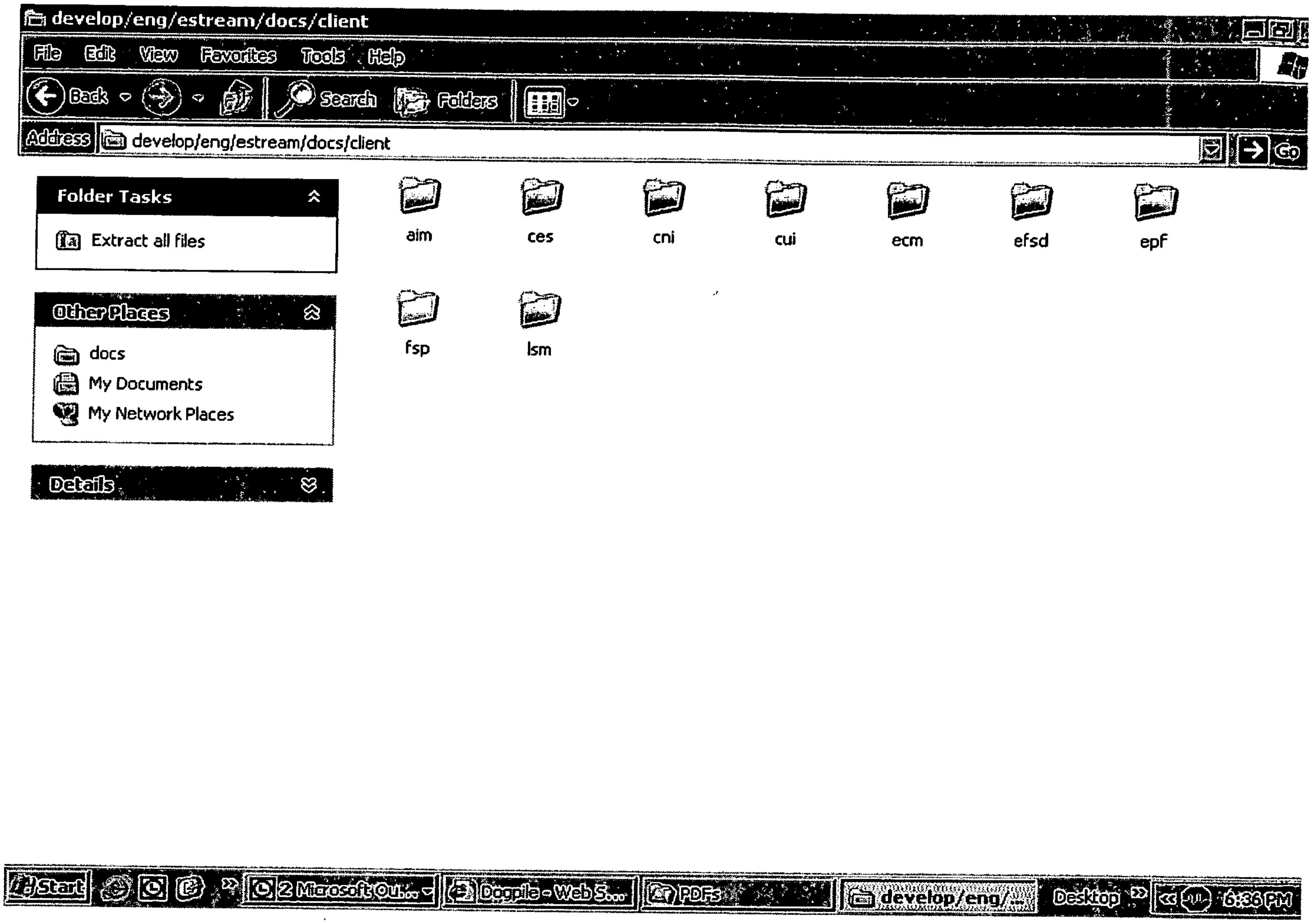
ECE may be deactivated automatically at eStream user logout [eSLogin.exe deactivate] or manually on demand via the CUI. When the ECE is deactivated, the following events are performed:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to stop eStream-related activities:
    - NoCluster: CES sends IOCTL to deactivate [or may continue active]
    - EFSD: CES sends IOCTL_EFS_SHUTDOWN_FS
    - FileSpoofer: CES sends IOCTL_FS_STOP_SPOOFING
- If any ECE routines need to execute separate termination on client deactivation, those routines are called at this point.

ECE brings up a browser to provide web-based ABOUT/HELP information for eStream and to allow the user to interact with his ASP to obtain/modify account-specific data. For eStream 1.0, this communication is handled via the standard mechanism used by currently-available windows application that provides web access, i.e., the user's default browser is invoked in a separate window with a particular URL and set of parameters [decided at 9/7 meeting of Lacky, Bhaven, Manoj, and me]. A common way to accomplish this is to execute a variant of the ShellExecute command.

Automatic Transition from READY to ACTIVE mode Plus Error Handling

If the eStream client software is not active when an eStream user tries to execute an eStream application, it is attractive to either give a meaningful error message or to automatically activate eStream. [To my knowledge, doing either is an eFSD issue; I am including a discussion of the issue here since the topic seems to be open at this time.] To give a meaningful error message, eFSD needs to see the relevant file reference, meaning that either the eStream drive letter needs to exist or eFSD needs to be registered as a UNC provider; otherwise, we may give a somewhat inscrutable error message. To automatically transition from READY to ACTIVE when we see a relevant file reference, we need to ensure that file spoofing is not needed prior to eFSD receiving the reference to the inactive eStream file system for apps of interest. Please note that it is not required that eStream client software be activated automatically if CBM finds it inactive.


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/client
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
2 Microsoft Out...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
6:36 PM

[/STANDARD.01]

eStream Client Networking Low Level Design

Dan Arai
Version1.6
[redacted]

Functionality

The Client Network Interface (CNI) provides the interfaces for sending messages to servers and provides threads for receiving responses and dispatching them appropriately. It uses the eStream Messaging Service (EMS) APIs to send and receive various messages to and from the application servers and SLiM servers.

The number of threads in the CNI will depend on the functionality available from the EMS. In particular, more threads are necessary if the EMS provides asynchronous messaging capability (and the CNI uses this interface). The interfaces presented by the CNI are identical for both cases, but the internal organization of the component is not.

The prefetcher will make calls to client networking interfaces (indirectly through ECM-ReservePage) to send requests for pages. Similarly, the LSM will make calls to acquire access tokens and subscription information.

The networking component is responsible for examining the stream of requests to it and deciding when to coalesce multiple page requests into a single request to the server.

The EMS does not provide reliability in the event of server failure. The CNI is responsible for handling server failover and reissuing failed requests on different servers. The CNI abstracts the servers from other parts of the system. Clients of the CNI don't need to specify a particular server to make a request.

Since the client networking component is where timeouts and retries occur, it is the component that controls the policies for how long we wait for a connection to time out and how many times we retry a request before giving up. These parameters will be tunable. Any other parameters of the CNI that make sense to tune will be tunable.

The CNI is also the component responsible for implementing the server selection policy.

Data type definitions

The CNI uses the request structure defined by the ECM.

The CNI maintains an internal queue of messages that must be sent to servers. This queue is not exposed outside of the CNI. Like the ECM request queue, this queue will be maintained as a circular, doubly-linked list.

```
typedef struct _NWRequest
{
     NWRequestType type;
     union {} parameters; /* params, depends on type */
     struct _NWRequest *next;
     struct _NWRequest *prev;
} NWRequest;

typedef enum
{
     CNI_PAGE_READ,
     CNI_ACQUIRE_ACCESS_TOKEN,
     CNI_GET_LATEST_APP_INFO,
     CNI_RENEW_ACCESS_TOKEN,
     CNI_RELEASE_ACCESS_TOKEN,
     CNI_REFRESH_APP_SERVER_SET,
     CNI_GET_SUBSCRIPTION_LIST
} NWRequestType;
```

The CNI provides an enumeration of the parameters that can be tuned. This enumeration is expected to grow as the number of tunable parameters grows.

```
typedef enum
{
     CNI_NUM_RETRIES,
     CNI_TIMEOUT,
     CNI_PROXY_ADDRESS,
     CNI_EFFECTIVE_BANDWIDTH
} NWTunableParameter;
```

Related Components

The prefetcher and LSM call on the CNI to send requests to the app and SLiM servers. The CNI makes calls to the ECM and LSM to inform them of responses that have come back from the server. The CNI will also make calls to EFSD interface functions when pages come back that satisfy EFSD requests.

Interface definitions

CNIGetPage

eStreamStatus CNIGetPage(
    IN ApplicationID *app*,
    IN EStreamPageNumber *page*
);

CNIGetPage is the interface used by the ECM function **ECMReservePage** to request that a page be sent by the server. (**ECMReservePage** is called indirectly by the pre-

fetcher.) Note that no distinction is made between prefetches and demand fetches. To prevent race conditions or deadlock, the requested pages must already be marked as "in flight" in the index, and any requests for these pages from the EFSD must already be on the "in flight" queue before calling this interface.

The CNI is responsible for selecting a server to direct this request to, and resending in the event of network or server failure. It will coalesce requests for multiple pages from the same application into a single request to the server.

CNIGetSubscriptionList

eStreamStatus CNIGetSubscriptionList(
    IN string *Username,*
    IN string *Password*
);

CNIGetSubscriptionList enqueues a request to acquire a subscription list from a SLiM server. When the subscription list is returned by the server, the client response thread will notify the LSM of the returned data via a callback defined in the LSM document.

CNIGetLatestApplicationInfo

eStreamStatus CNIGetLatestApplicationInfo(
    IN uint128 *SubscriptionID*
);

CNIGetLatestApplication enqueues a request to get the latest application information for a particular app. When the server returns the result, the CNI will notify the LSM of the returned data via a callback defined in the LSM document.

CNIAcquireAccessToken

eStreamStatus CNIAcquireAccessToken(
    IN uint128 *SubscriptionID,*
    IN string *Username,*
    IN string *Password*
);

CNIAcquireAccessToken will cause the CNI to contact a SLiM server to retrieve an access token. The CNI is responsible for issuing retries if no response is received for a request. The CNI will call the appropriate LSM callback function when the data come back.

CNIRenewAccessToken

eStreamStatus CNIRenewAccessToken(
    IN AccessToken *Token,*
    IN string *Username,*
    IN string *Password*
);

CNIRenewAccessToken will enqueue a request for access token renewal. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIReleaseAccessToken

eStreamStatus CNIReleaseAccessToken(
    IN AccessToken *Token,*
    IN string *Username,*
    IN string *Password*
);

CNIReleaseAccessToken will enqueue a request for releasing an access token. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIRefreshAppServerSet

eStreamStatus CNIRefreshAppServer(
    IN AccessToken *Token,*
    IN uint32 *BadQOS,*
    IN uint32 *NoService*
);

CNIRefreshAppServerSet will enqueue a request for refreshing the app server set. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

The client networking component will also have routines for getting and setting tunable parameters.

CNISetParameter

eStreamStatus CNISetParameter(
    IN NWTunableParameter *type,*
    IN void **value*
);

CNISetParameter sets a parameter. The actual type of *value* is determined by *type.*

CNIGetParameter

eStreamStatus CNIGetParameter(
    IN NWTunableParameter *type,*
    OUT void **value*
);

CNIGetParameter queries the current value of a parameter. The actual type of *value* is determined by *type.*

Component design

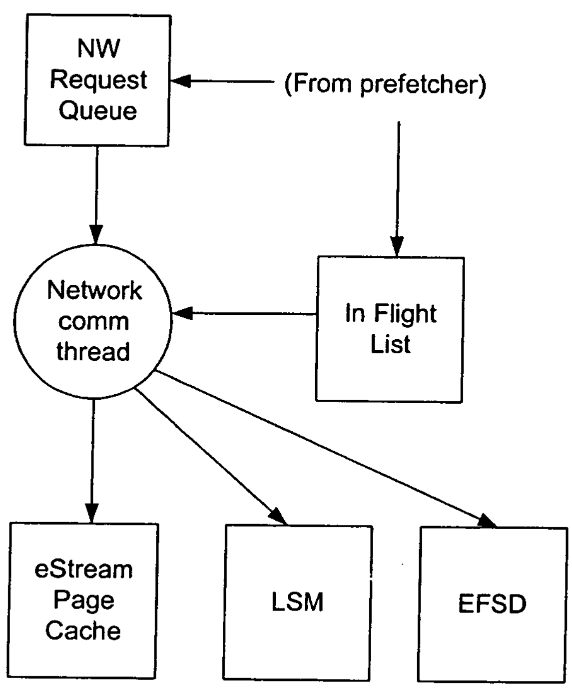
The internal organization of the client networking depends on the mechanisms available from EMS. Internally, the CNI interface functions put requests on a queue, and one or more threads services these requests by using the EMS to send messages to servers.

Synchronous Server Calls

If EMS only provides a synchronous messaging service, a single thread will be used to perform all necessary actions. The CNI interfaces will put appropriate requests on the network request queue. They will also wake up the network communication thread, if necessary.

The network communication thread's job is relatively simple. When it wakes up, it performs the following tasks:
- choose a set of requests to be coalesced and remove these from the request queue
- retrieve a server set via LSMGetAppServerSet or LSMGetSLiMServerSet, and choose a particular server for this request
- make a synchronous EMS call to send the request
- dispatch the response to the appropriate LSM or ECM callback

If the synchronous messaging mechanism becomes a performance bottleneck, we can have multiple network communication threads to increase concurrency.



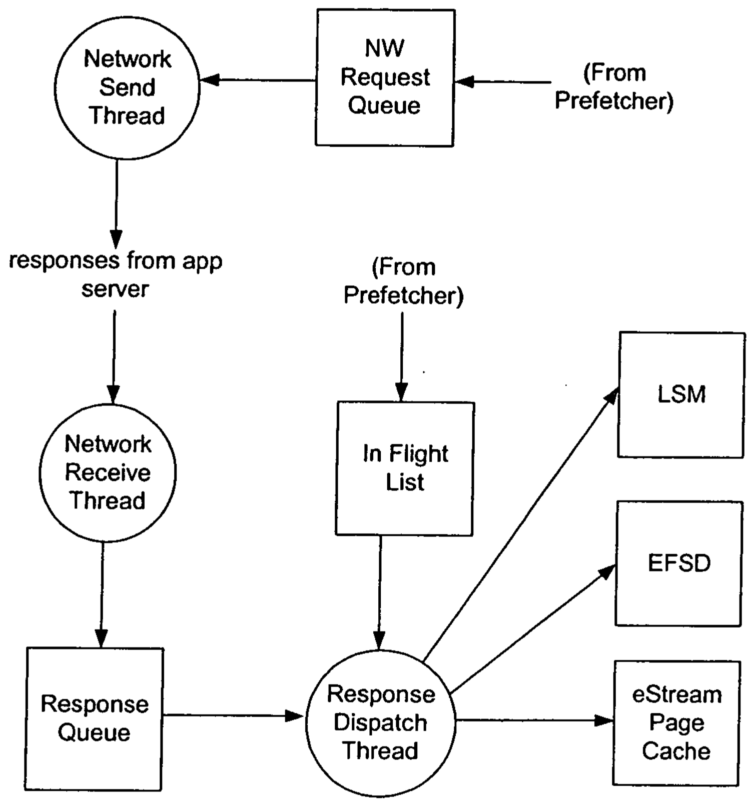
Asynchronous Server Calls

The asynchronous case is a little bit more complex. Because of the proposed asynchronous call architecture, the client NW requires three threads. The CNI interfaces work just as they do in the synchronous case. They put requests on the network request queue, and wake up the network send thread. However, the actions performed by the CNI's worker threads differ in the asynchronous model.

The network send thread is periodically awoken, and it coalesces requests off the NW request queue and sends them to the server. Unlike in the synchronous model, this thread does not synchronously wait for the request to come back from the server. Instead, it simply sends requests until the queue is empty, then goes back to sleep.

The network receive thread waits for responses to come back from any server. Because of the EMS's asynchronous call implementation details, this thread posts returned data to a queue of responses to be handled by another thread. The network receive thread is also responsible for handling timeouts and reissuing those network requests on different servers.

Finally, the response dispatch thread pulls responses off the response queue, and handles the work of dispatching them appropriately.



Handling Network Failure
When the client networking component is notified of a message failure by the EMS, the client worker thread will attempt to reissue the request on a different server.

Coalescing Multiple Requests
The CNI will coalesce multiple page requests that come from the LSM into a single request to an application server. Multiple pages requests for the same application may be coalesced. No other types of requests may be coalesced, including page requests for dif-

ferent applcations. The CNI will not produce requests larger than the maximum allowed by the application server.

Handling Persistent Failures
There will be some persistent failures that will result in the network being unable to fulfill page requests in a timely fashion. This may be due to network or server failure. (These may be indistinguishable from the CNI's point of view.) When the CNI has failed to satisfy a request for a certain amount of time, it will need to ask the user if he wants it to continue retrying, or if it should let the application terminate. It will do this via the **CUIAskUserYesNo()** interface. The client software control panel should include an option to always wait until the server is available, and never ask the user if he wants the application terminated.

Testing design

Unit testing plans

The testing harness for the networking component will be a set of dummy EMS drivers and a dummy NW client. The dummy EMS driver will be capable of performing a variety of actions, including returning appropriate responses, returning inappropriate responses, and timing out without any response. The dummy NW client will have knowledge about the expected EMS behavior, and will verify that the data it gets back from the network component are as expected.

Stress testing plans

Failure testing plans

The client NW is the sole component responsible for implementing server failover. In order to test this code, it is necessary to implement a server with predefined bad behavior. The server failure modes that must be tested include
- server that accepts a connection on a socket but doesn't respond to any requests
- server that closes the socket before sending a response
- server that closes the socket in the middle of a response
- server that sends a partial response and then just stops
- server that satisfies n requests then closes the socket or refuses to service more

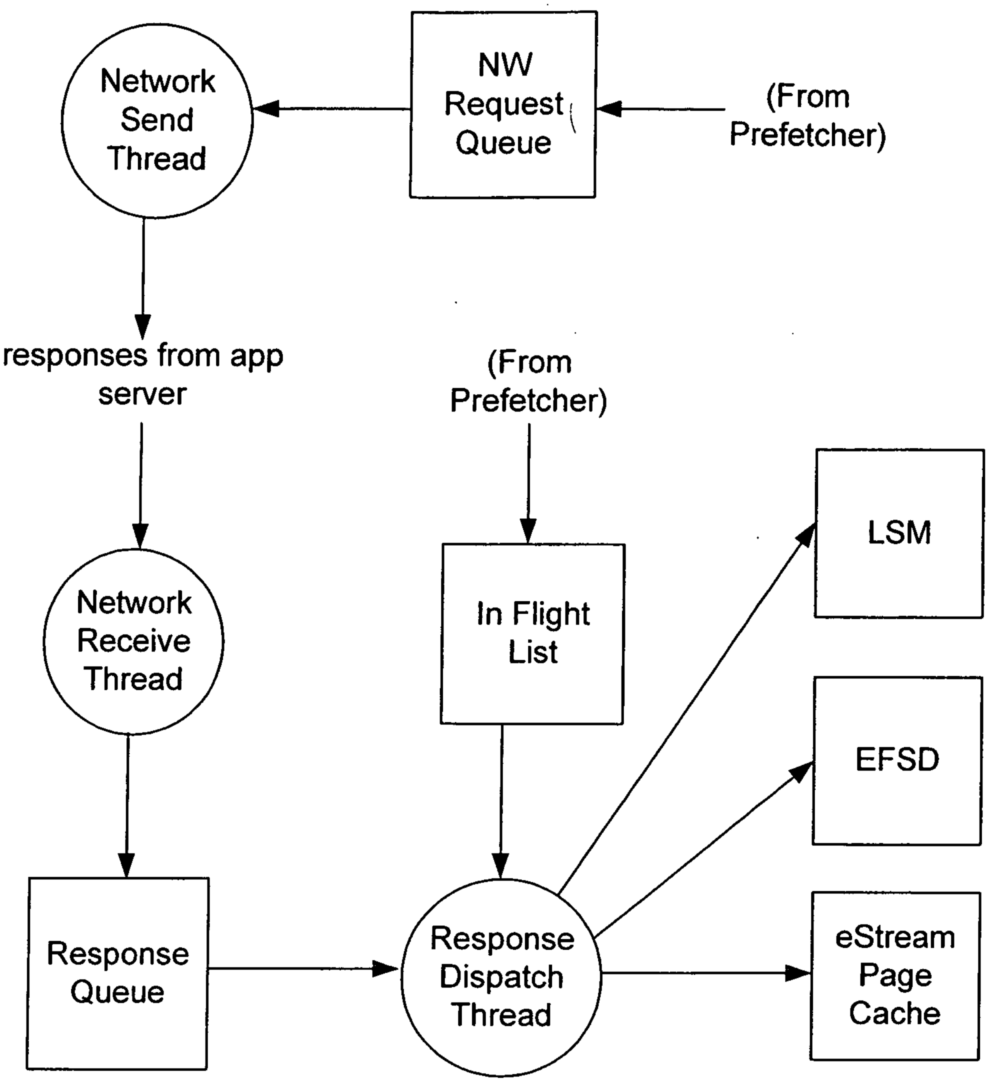
It is important that we cover scenarios that look like network failures and ones that look like server failures. (Are there other failure modes that are interesting?)

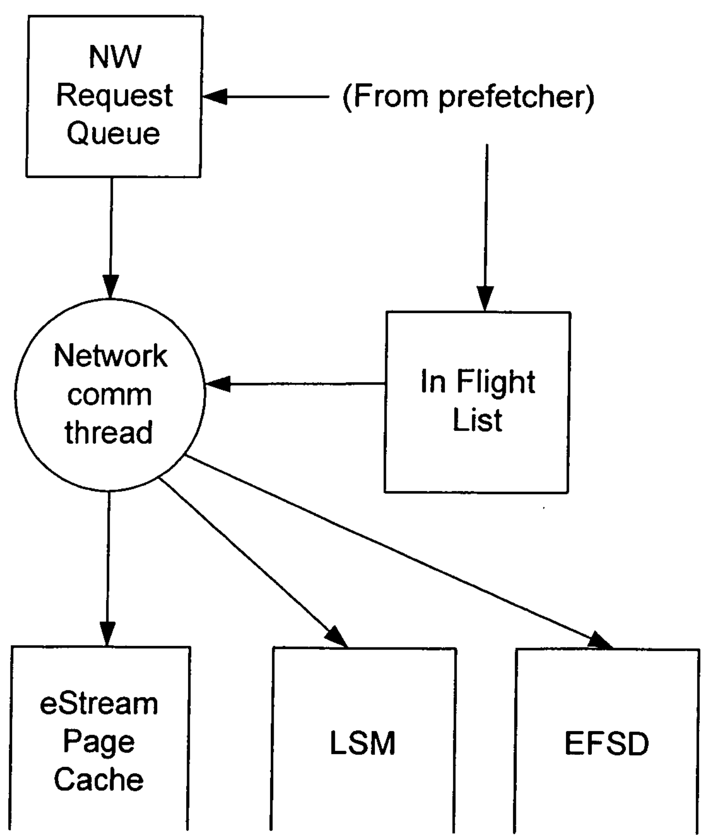
Cross-component testing plans

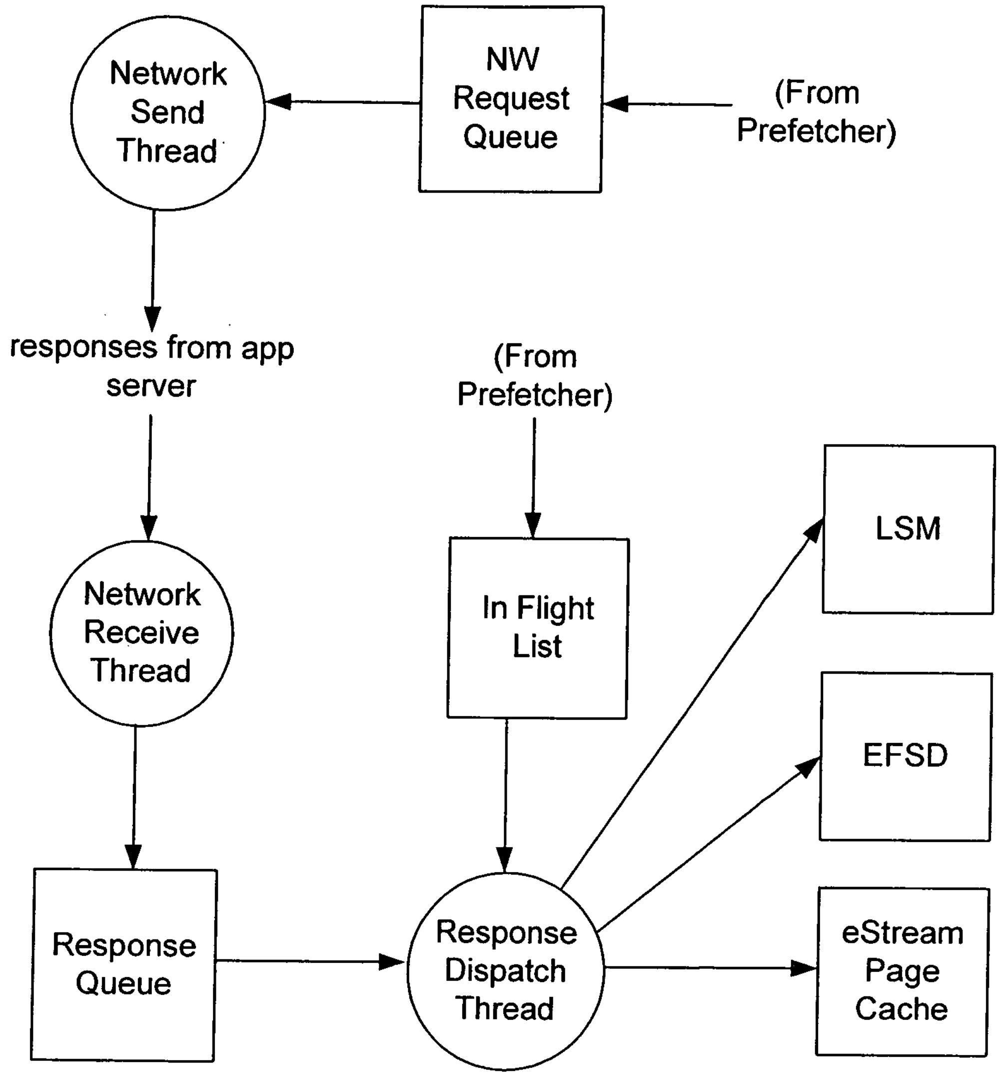
Cross-component testing of the client NW includes integration testing with the EMS, the LSM, and the prefetcher. Testing with the EMS can be performed in a manner similar to unit testing in conjunction with a specially written server. Testing with the LSM or pre-

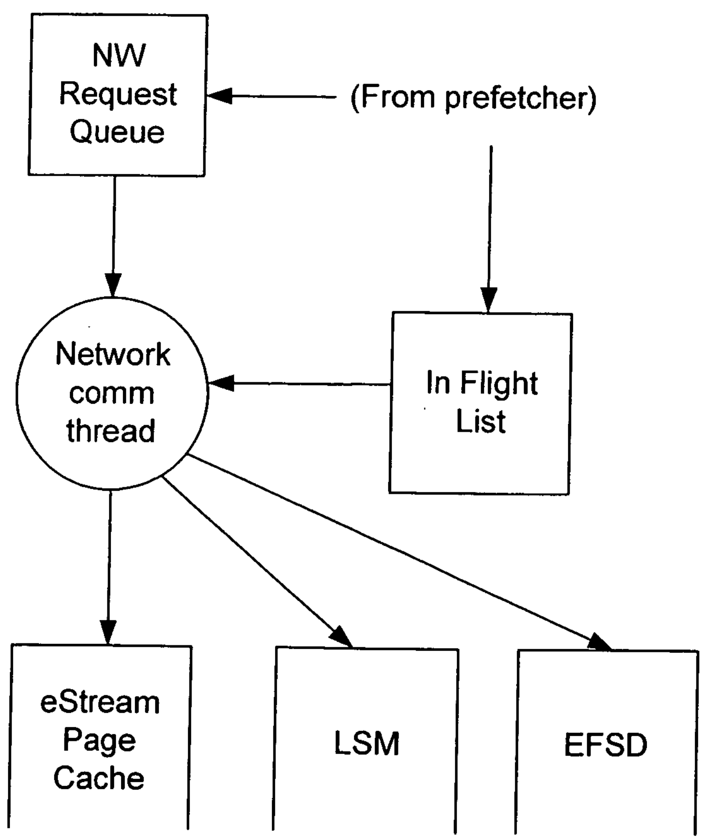
fetcher can be performed in isolation by writing drivers for either the LSM or prefetcher, and using a dummy or real EMS. I'm not sure if this sort of testing is worth the effort to write the appropriate harnesses. Verifying the output of such a combined system is certainly trickier than testing any component in isolation.

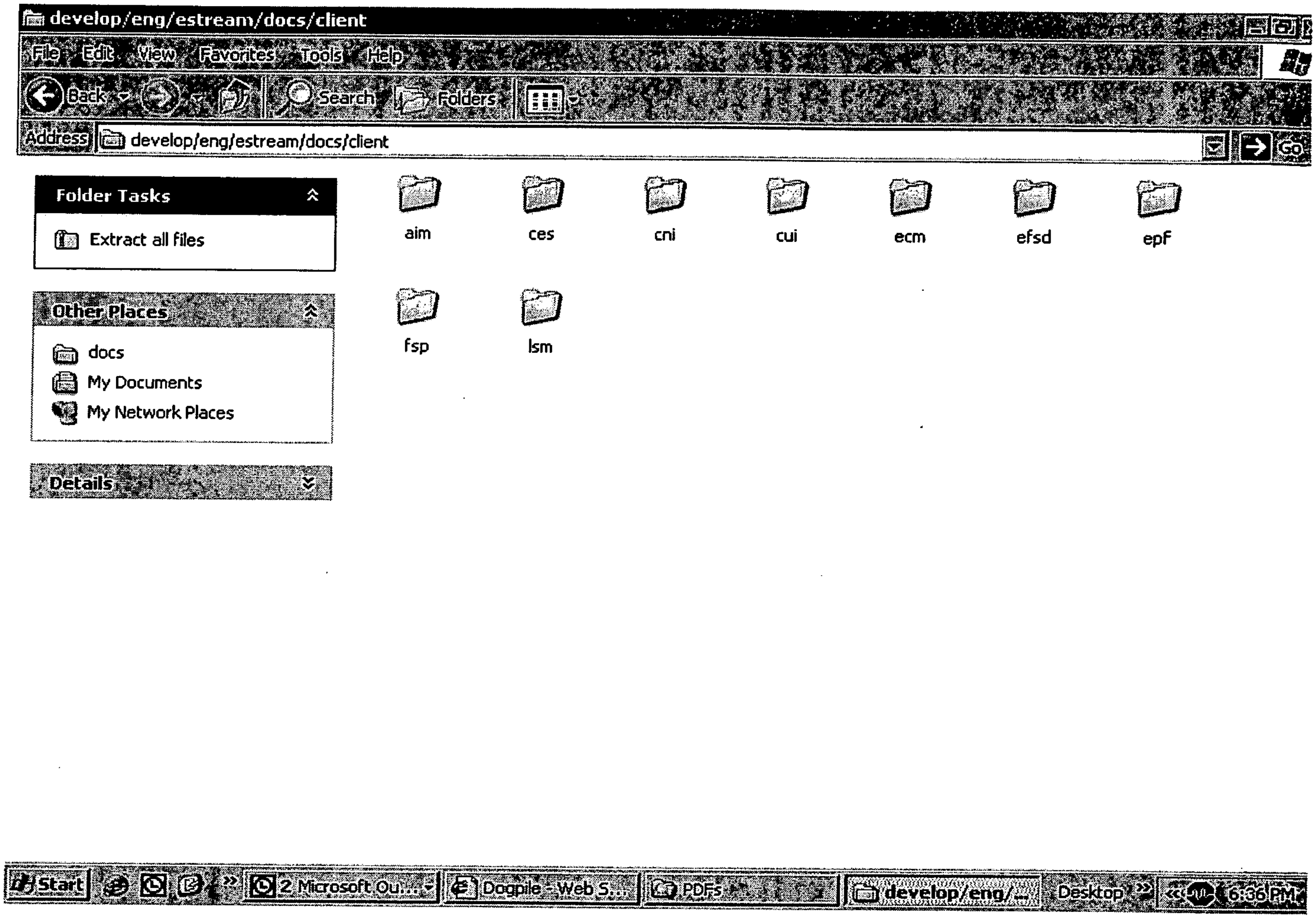
Open Issues


Network Send Thread
NW Request Queue
(From Prefetcher)
responses from app server
(From Prefetcher)
LSM
In Flight List
Network Receive Thread
EFSD
Response Queue
Response Dispatch Thread
eStream Page Cache


NW Request Queue
(From prefetcher)
Network comm thread
In Flight List
eStream Page Cache
LSM
EFSD


Network Send Thread
NW Request Queue
(From Prefetcher)
responses from app server
(From Prefetcher)
LSM
In Flight List
Network Receive Thread
EFSD
Response Queue
Response Dispatch Thread
eStream Page Cache


NW Request Queue
(From prefetcher)
Network comm thread
In Flight List
eStream Page Cache
LSM
EFSD



[/STANDARD.01] [redacted]

eStream 1.0 Client User Interface Low Level Design

Anne Holler *[redacted]* *Version 2.3*

Functionality

This document represents the low level design of the Client eStream User Interface [CUI], an eStream client module that:

- supports asynchronous communication from various eStream client components to the user, optionally soliciting responses
- provides a visual indication to the user that the eStream client software is running, via the presence of the eStream systray icon
- displays ABOUT and HELP information to the eStream client user, including a link to a web site with FAQs and access to support with respect to eStream
- interfaces with a (planned but not yet designed) client ASP user interface, which allows the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT and HELP information, etc, from the client without requiring that the user manually point his browser at the ASP website and login
- allows an eStream user to request a variety of [optional/advanced] eStream client management functions (described below)

Please note that the eStream CUI is not used to launch eStream applications; eStream applications are launched (like locally-installed applications) via Start/Programs or via desktop icons. The eStream CUI is considered to be a utility interface (in Windows UI parlance) and will follow the Windows UI conventions of this category of program.

Data definitions

eStream client software will be developed in a manner which facilitates porting its user messages and interfaces to languages other than English. Since the CUI is responsible for displaying messages and help text to client users, it is the natural module to "own" the issue of localization.

Windows Resource Files will be used for all user messages; CUI's callers will pass handles to messages they wish displayed. All user displays/menus/bitmaps will be developed in a manner consistent with the guidelines outlined in "Developing International Software for Windows 95 and Windows NT" by Nadine Kano [MSDN Online Library]. Please note that eStream 1.0 is targeted to the English language market, and translation of the Resource Files to other languages is not planned to be completed prior to first product shipment.

The AppName, VersionName, & Message info supplied in the eStreamAppInfo are in the same language as the version of the app (i.e., Japanese Word => Japanese).

Interface definitions

Client Component Acronyms

- CES: Client EStream Startup
- CNI: Client Network Interface
- CUI: Client User Interface
- ECM: EStream Cache Manager
- EPF: EStream PreFetch/Fetch
- LSM: License Subscription Manager

From <various client components> to CUI:

- **CUIInformUser**(IN Message, IN OptionalHelpInfo)
- **CUIAskUserYesNo**(IN Message, IN OptionalHelpInfo, IN CheckAskAgain, OUT Response, OUT DontAskAgain)
- **CUIAskUserPassword**(IN Message, OUT Response, OUT Retain)

From CUI to CES:

- CESSetActivate(IN Boolean ActivateAtLogin)
- CESDeactivate(VOID)

From CUI to LSM:

- LSMGetAspList(OUT NumAsps, OUT AspID[])
- LSMGetAspInfo(IN AspID, OUT ASPWebServerName, OUT UserName)
- LSMUpdateAllSubscriptionStatus(VOID)
- LSMGetAppList(IN AspID, OUT NumApps, OUT AppID[])
- LSMGetAppInfo(IN AppID, OUT AppName, OUT VersionName, OUT Message, OUT AppInstallStatus, OUT AppUpgradeStatus)
- LSMInstall(IN AppID, OUT InstallStatus)
- LSMUninstall(IN AppID, OUT UninstallStatus)
- LSMUpgrade(IN AppID, OUT UpgradeStatus)

From CUI to ECM:

- ECMSetCacheInfo -- causes ECM to recheck registry for cache size
- ECMGetCacheInfo

From CUI to EPF:

- EPFPrefetchSet(IN OnOff)

From CUI to CNI:

- CNIGetParameter(IN ParmName, OUT ParmValue)
- CNISetParameter(IN ParmName, IN ParmValue)

Component design

Asynchronous Messages to eStream User

There are several (hopefully rare) situations that arise asynchronously, about which components of the eStream client software inform the user. In some cases, these situations necessitate user response. CUI delivers asynchronous messages via pop-up dialog boxes. If no user input is needed, other than confirmation that the user has seen the message, the box displays text with a simple Continue button [**CUIInformUser**]. If the message is a question to which a yes/no response from the user is sought, the pop-up dialog box contains Yes and No buttons with the recommended selection emphasized [**CUIAskUserYesNo**]. If the message requires that the user enter a password, the pop-up contains an edit box in which character echoing is shielded [**CUIAskUserPassword**]. Help buttons with supporting amplifying passages are supplied in situations in which extra information can aid user understanding.

eStream Systray Icon

EStream client software can be activated automatically at user login, manually via a Start/Program entry, and automatically when an EFSD request arrives and EFSD is inactive. [Please see the CES design materials for more information on activation.] The eStream client software must be active for the user to execute eStream applications; the presence of the systray icon indicates that the software is active. Right-clicking the icon engenders a pop-up menu from which the user can reach about/help info, ASP client account interfaces, and eStream client management functions.

EStream About/Help Information

The chief external interface involved in eStream-specific ABOUT+HELP is launching a browser and passing arguments to it. The user's default browser is used for this. Please note that the user's default browser must be one of the browsers specified as required for eStream 1.0 (IE4 or higher, Navigator 4 or higher).

ASP Client Account Interfaces

It is beyond the scope of this document to describe the interfaces with a (planned but not yet designed) client ASP user interface, which allows the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT+HELP information, etc., from the client without requiring that the user manually point his browser at the ASP website and login. However, the CUI implementation includes bringing up a browser and passing arguments to it, which is key to supporting the success of this future functionality.

EStream Client Management Functions

The typical user should not normally need to use the eStream client management functions available from the CUI. Subscription information is synchronized automatically, applications are installed/uninstalled when the subscription information is synched, eStream client software is activated at login & deactivated at logout, the cache is right-sized, etc. The functions included in this section are supplied for use in special circumstances and/or in support situations.

eStream client management functions include the following:

- For ASP accounts known to this client:
  - View account information **[LSMGetAspList, LSMGetAspInfo]**
  - Request synchronization of application subscription information [handled automatically at client eStream activation, included here for flexibility] **[LSMUpdateAllSubscriptionStatus]**
- For application subscriptions known to this client:
  - View subscription information [this may engender a synch] **[LSMGetAppList, LSMGetAppInfo]**
  - Install subscribed application [handled automatically at synch time, included here for flexibility] **[LSMInstall]**
  - Uninstall application [handled automatically at synch time, included here for flexibility] **[LSMUninstall]**
  - Upgrade application to latest version [handled automatically, flexibility] [**LSMUpgrade**]
- For eStream client software:
  - Specify automatic activation at login vs manual activation via program/start [it is proposed that we assume the former is always the case] **[CESSetActivate]**
  - Any general user preference wrt terminating app for very slow network response (always wait vs. terminate if too slow)
- For eStream client cache [primarily for users who do not allow eStream to right-size their cache]:
  - Display current size and utilization [obtained from registry, **ECMGetCacheInfo**]
  - Set size: allow eStream to right-size vs. manually increase/decrease size [**ECMSetCacheInfo**]
  - Disable/enable prefetching **[EPFPrefetchSet]**
- For eStream client network interface:
  - Display recent eStream effective bandwidth [possibly displayed as hover-over on eStream systray icon] **[CNIGetParameter]**
  - View/Set proxy IP address [may use info from control panel internet options widget] **[CNIGetParameter, CNISetParameter]**
  - View/Set connection time-out value **[CNIGet/SetParameter]**
  - View/Set number retries value **[CNIGet/SetParameter]**
- Deactivate eStream [**CESDeactivate**]

Testing design

Unit/Coverage testing plans

For the eStream client management functions and the ABOUT+HELP displays, a UI scripting tool like Rational's Visual Test will be used to automate testing. Unit testing will involve creating driver keystrokes that exercise all menu selections and stubs of other client modules that will report as many asynchronous error conditions as possible. Using Rational's PureCov tool, PFA coverage should be as close to 100% as possible.

Cross-component testing plans

With respect to the eStream client management functions, the CUI has many interfaces with LSM, and hooking the two together is a win-win in terms of facilitating the testing of both; this combination will be the first cross-component integration. CUI's unit test scripts are expected to continue to be useful in driving the testing of the CUI/LSM combination. Other client management interfaces will be added and tested as the associated components are completed, with continued emphasis on ensuring that asynchronous error messages are provoked.

Stress testing plans

Need/strategy unclear.

Upgrading/Supportability/Deployment design

A chief motivation for many of the user interfaces in the CUI design, particularly in the eStream client management functions area, is user support and deployment support.

Implementation Issues

- Investigate obtaining ProxyIPAddress from control panel internet options widget. [BTW, why aren't our competitors doing this?]
- Some items accessed by CUI are per-user (ASP, applications, activation) and some items are per-client (cache size, network interface). We need to make sure this is clearly communicated in the UI.
- API needs to be available so that every button/widget can be manipulated programmatically to facilitate testing.

eStream 1.0 Client eStream User Interface Straw Man

Anne Holler *[redacted]* ** Version 1.3*

Introduction

This document presents background information related to the Client eStream User Interface [CUI], an eStream client module that:

- supports asynchronous communication from various eStream client components to the user, optionally soliciting responses
- provides a visual indication to the user that the eStream client software is running, via the presence of the eStream systray icon
- displays ABOUT and HELP information to the eStream client user, including a link to a web site with FAQs and access to support with respect to eStream
- interfaces with a (planned but not yet designed) client ASP user interface, which would allow the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT and HELP information, etc, from the client without requiring that the user manually point his browser at the ASP website and login
- allows an eStream user to request a variety of [optional/advanced] eStream client management functions (described below)

The document is intended to provide a baseline for discussions prior to proceeding with a detailed low-level design.

Please note that the eStream CUI is not used to launch eStream applications; eStream applications are launched (like locally-installed applications) via Start/Programs or via desktop icons. The eStream CUI is considered to be a utility interface (in Windows UI parlance) and will follow the Windows UI conventions of this category of program.

Asynchronous Messages to eStream User

There are several (hopefully rare) situations that arise asynchronously, about which components of the eStream client software will inform the user. In some cases, these situations necessitate some user response. CUI delivers asynchronous messages via pop-up dialog boxes. If no user input is needed, other than confirmation that the user has seen the message, the box will display the text along with a simple Continue button [**CUIInformUser**]. If the message is a question to which a yes/no response from the user is sought, the pop-up dialog box will contain Yes and No buttons with the recommended selection emphasized [**CUIAskUserYesNo**]. If the message requires that the user enter a password, the pop-up will contain an edit box in which character echoing is shielded [**CUIAskUserPassword**]. Help buttons with supporting amplifying passages are supplied in situations in which extra information can aid user understanding.

Please note that the eStream client software is expected to be developed in a manner which facilitates porting its user messages and interfaces to languages other than English. Hence, messages are to be obtained from a catalog or resource file to ease translation.

eStream Systray Icon

The eStream client software must be active for the user to execute eStream applications; the presence of the systray icon indicates that it is active. Right-clicking the icon engenders a pop-up menu from which the user can reach help/about info, ASP client account interfaces, and eStream client management functions.

EStream About/Help Information

The chief external interface involved in eStream-specific ABOUT+HELP is launching a browser and passing arguments to it. It is expected that the browser utilized for this feature will be Internet Explorer; if so, we will require that Internet Explorer not be uninstalled from the user's windows platform (which is difficult to do, as demonstrated at Microsoft's antitrust trial). Please note that the user can still employNetscape Navigator to connect to the ASP, as the eStream 1.0 requirements document specifies; this is a separate issue of what browser is used for the client pass-through to ASP account functionality.

ASP Client Account Interfaces

It is beyond the scope of this document to describe the interfaces with a (planned but not yet designed) client ASP user interface, which weill allow the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT+HELP information, etc, from the client without requiring that the user manually point his browser at the ASP website and login. However, the CUI implementation includes bringing up a browser and passing arguments to it, which is key to supporting the success of this future functionality.

EStream Client Management Functions

The typical user should not normally need to use the eStream client management functions available from the CUI. Subscription information is synchronized automatically, applications are installed and uninstalled when the subscription information is synched, eStream client software is activated at login, the cache is right-sized, etc. The functions included in this section are supplied for special circumstances and/or support situations.

eStream client management functions include the following:

- For ASP accounts known to this client:
  - View account information **[LSMGetAspList, LSMGetAspInfo]**
  - Request synchronization of application subscription information [handled automatically at client eStream activation, included here for flexibility] **[LSMUpdateAllSubscriptionStatus]**
- For application subscriptions known to this client:
  - View subscription information [this may engender a synch] **[LSMGetAppList, LSMGetAppInfo]**
  - Install subscribed application [handled automatically at synch time, included here for flexibility] **[LSMInstall]**
  - Uninstall application [handled automatically at synch time, included here for flexibility] **[LSMUninstall]**
  - Upgrade application to latest version [handled automatically, flexibility] **[LSMUpgrade]**
- For eStream client software:
  - Specify automatic activation at login vs manual activation via program/start [it is proposed that we assume the former is always the case] **[CESSetStartup]**
- For eStream client cache [primarily for users who do not allow eStream to right-size their cache]:
  - Display current size and utilization [obtained from registry]
  - Set size: allow eStream to right-size vs. manually increase/decrease size **[ECMSetCacheSize]**
  - Disable/enable prefetching **[EPFPrefetchSet]**
- For eStream client network interface:
  - Display recent eStream effective bandwidth [possibly displayed as hover-over on eStream systray icon] **[CNIGetEffectiveBandwidth]**
  - View/Set proxy IP address [may use info from control panel internet options widget] **[CNIGetProxyIPAddress, CNISetProxyIPAddress]**
- Deactivate eStream **[CESShutdown]**

CUI Interfaces To/From Client Components

Client Component Acronyms

CES: Client EStream Startup
CNI: Client Network Interface
CUI: Client User Interface
ECM: EStream Cache Manager
EPF: EStream PreFetch
LSM: License Subscription Manager

From CUI to CES:

CESSetStartup(IN Boolean ActivateAtLogin)
CESShutdown(VOID)

From CUI to LSM:
LSMGetAspList(OUT NumAsps, OUT AspID[])
LSMGetAspInfo(IN AspID, OUT ASPWebServerName, OUT UserName)
LSMUpdateAllSubscriptionStatus(VOID)
LSMGetAppList(IN AspID, OUT NumApps, OUT AppID[])
LSMGetAppInfo(IN AppID, OUT AppName, OUT VersionName, OUT Message,
OUT AppInstallStatus, OUT AppUpgradeStatus)
LSMInstall(IN AppID, OUT InstallStatus)
LSMUninstall(IN AppID, OUT DeinstallStatus)
LSMUpgrade(IN AppID, OUT UpgradeStatus)

From CUI to ECM:
ECMSetCacheSize(VOID) -- causes ECM to recheck registry for updated cache size

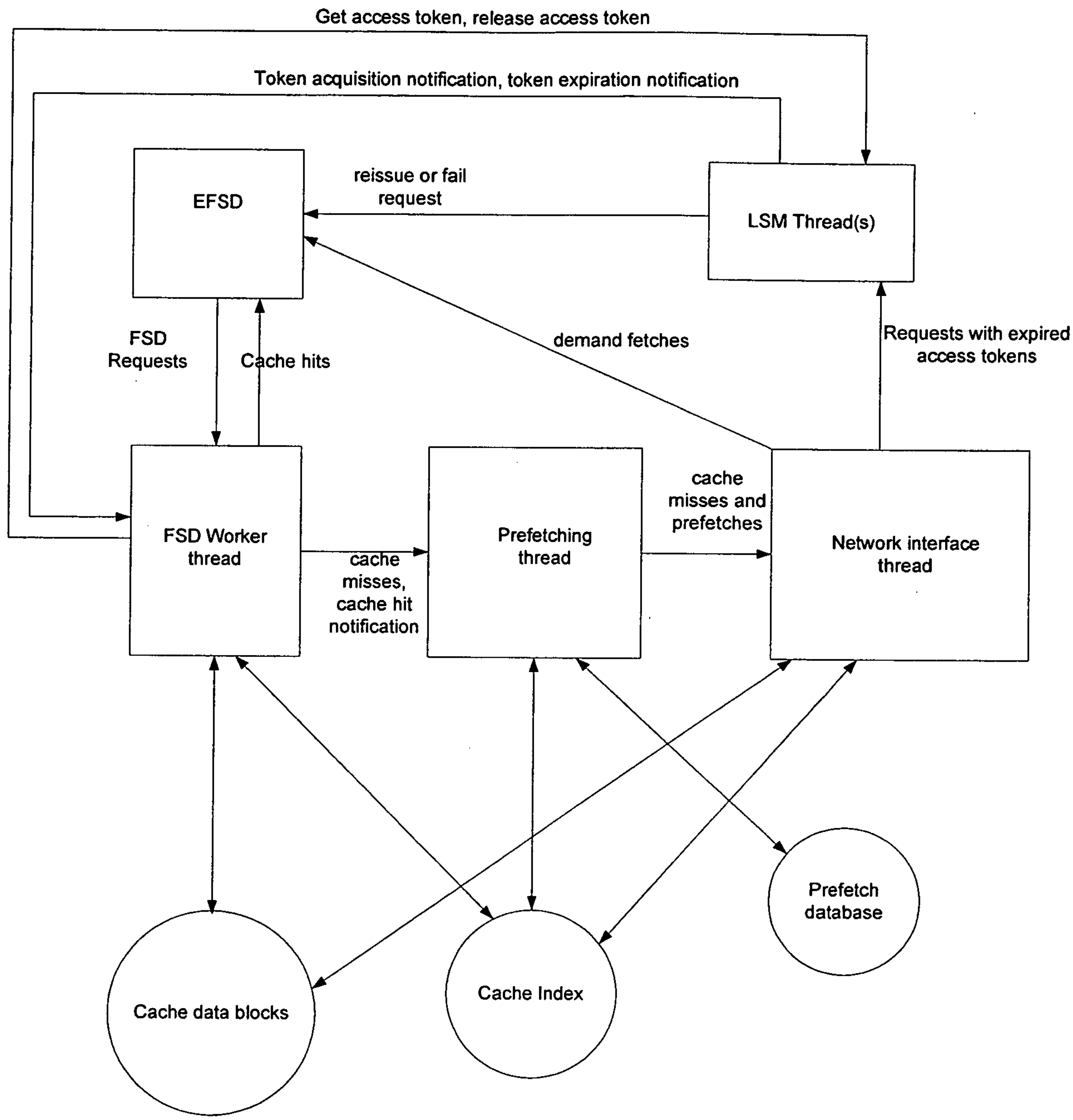
From CUI to EPF:
EPFPrefetchSet(IN OnOff)

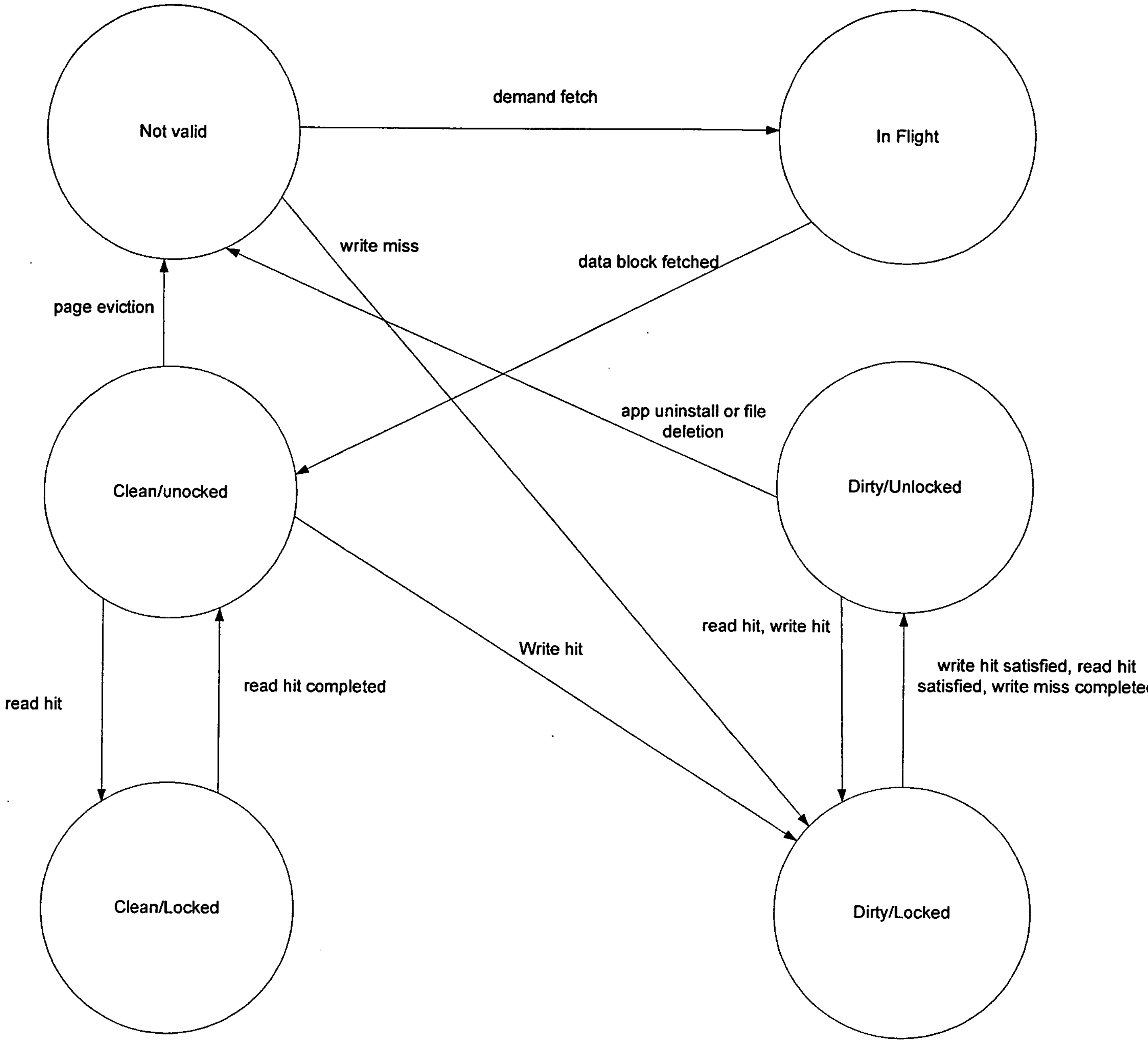
From CUI to CNI:
CNIGetProxyIPAddress(OUT ProxyIPAddress)
CNISetProxyIPAddress(IN ProxyIPAddress)
CNIGetEffectiveBandwidth(OUT EstreamRecentEffectiveBandwidth)

From <various client components> to CUI:
CUIInformUser(IN Message, IN OptionalHelpInfo)
CUIAskUserYesNo(IN Message, IN OptionalHelpInfo, IN CheckAskAgain,
OUT Response, OUT DontAskAgain)
CUIAskUserPassword(IN Message, OUT Response, OUT Retain)

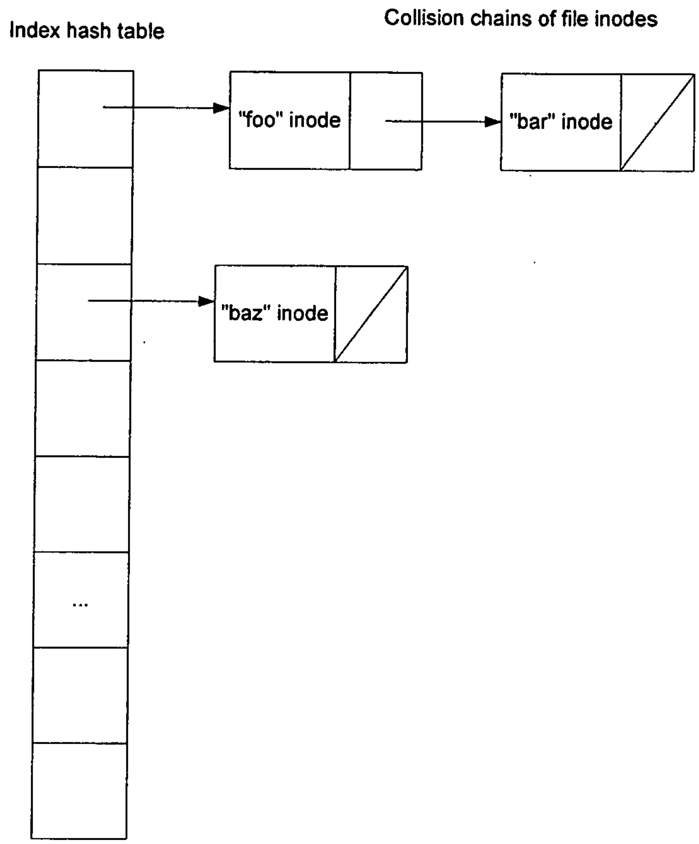
Issues

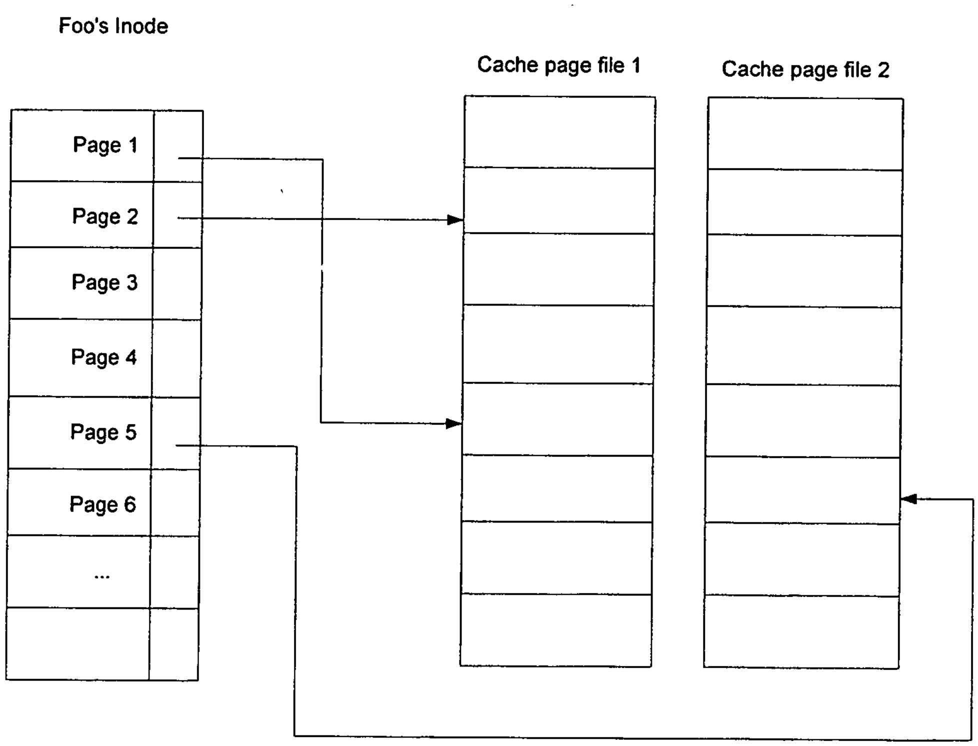
- Resolve general issue of whether eStream client management functions should be included in product at all (opinions on both sides).
- Within client management functions, should prefetch enable/disable be included?
- Should CNI get ProxyIPAddress from control panel internet options widget and should we therefore remove this menu item & interface from CUI?
- Will cache utilization be posted to the registry or available somehow?
- What is the interface to obtain recent effective eStream network bandwidth?

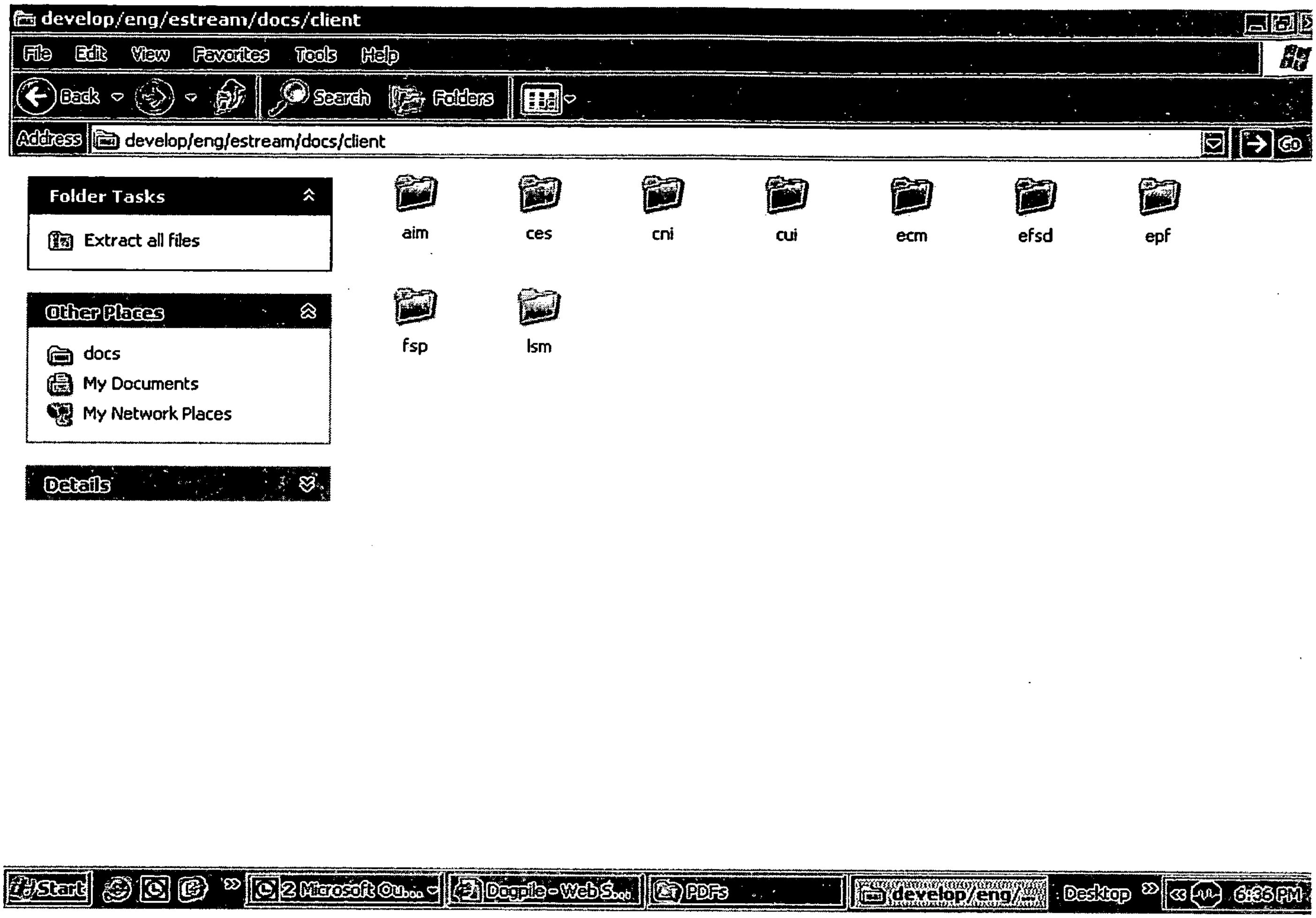

Get access token, release access token
Token acquisition notification, token expiration notification
EFSD
reissue or fail request
LSM Thread(s)
FSD Requests
Cache hits
demand fetches
Requests with expired access tokens
FSD Worker thread
cache misses, cache hit notification
Prefetching thread
cache misses and prefetches
Network interface thread
Prefetch database
Cache data blocks
Cache Index



Not valid
demand fetch
In Flight
write miss
data block fetched
page eviction
app uninstall or file deletion
Clean/unocked
Dirty/Unlocked
read hit, write hit
Write hit
write hit satisfied, read hit satisfied, write miss completed
read hit
read hit completed
Clean/Locked
Dirty/Locked


Index hash table
Collision chains of file inodes
"foo" inode
"bar" inode
"baz" inode
...


Foo's Inode
Cache page file 1
Cache page file 2
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
...



[/STANDARD.01] [redacted]

eStream 1.0 Cache Manager Low Level Design

Version 1.4

[redacted]

Omnishift Technologies, Inc.
Company Confidential

Functionality

The eStream cache manager implements much of the client-side functionality for handling the eStream file system. The cache manager handles all file system requests made by the operating system by reading information from the cache or by passing the requests along to the profiling and prefetching component to fetch missing data from the network.

The cache manager will initially be implemented in user space, but it may be useful to migrate it to the kernel for improved performance. In user space, it will be part of the eStream client process. In the kernel, it will probably be a device driver distinct from the eStream file system driver.

The cache manager manages the on-disk cache of file system data, and the in-memory data structures for managing this cache. It does not manage prefetching of data from the server; that is the role of the eStream Profiling and Fetching (EPF) component. A separate networking component handles the network traffic. This component will also be described separately.

Since there is no overall discussion of the client architecture at a more detailed level than the high level design, this document will cover that as well.

Multiple cache page files will be supported. Each cache page file may be up to 2 GB in size. Different cache files may reside on different or the same logical disk (i.e. Windows drive letter.)

Data type definitions

An application ID uniquely identifies an eStream application. Just what constitutes "one" eStream application is not entirely defined, but different "builds" of the "same" app will be considered different eStream applications. For example, the Chinese-language version of Office is a different eStream application than the English-language version.

```
typedef uint128 ApplicationID;
```

The eStream page number is the data type used to describe a page number within a particular file. Note that this is a page offset, not a byte offset. For eStream 1.0, the cache manager will only support 2 GB cache files.

```
typedef uint32 EStreamPageNumber;
```

The fileId is used to uniquely identify a file within the universe of all eStream files across all eStream applications.

```
typedef struct {
     ApplicationID App,
     int32 File
} fileId;
```

The eStream page size is the fundamental size for eStream requests. This size is in bytes.

```
#define ESTREAM_PAGE_SIZE 4096
```

The eStream file system uses the file time format of the Windows operating system. If the client runs on a system with a different native time format, the client software will be responsible for translating between the native format and the eStream format. The Windows data format is a 64-bit counter of the number of 100-nanosecond periods since January 1, 1601.

EStream metadata is the file information supported by the eStream file system. This metadata is independent of the client or server operating system.

```
typedef struct
{
     uint64 CreationTime;
     uint64 AccessTime;
     uint32 FileSize;
     uint32 FileSystemAttributes;
     uint32 EStreamAttributes;
} Metadata;
```

The eStream inode contains the layout of a file in the cache. Each inode has the following structure:

```
typedef struct
{
     FileId Id; /* ID of this file; search parent for
name*/
     Metadata Metadata;
     FileID Parent; /* parent directory's file id */
     uint32 NumPages;
     PageInfo *Pages;
} EStreamInode;
```

The PageInfo array is variable sized. There is one entry in the pages array for each page in the file (not for each page cached, since we need to know whether the pages are present or not...) Note that the inode is only used in the "robust" implementation.

```
typedef struct
{
     EStreamPageNumber CachePageNumber;
     PageStatus Status;
     unsigned char Priority;
     PageChecksum Checksum;
} PageInfo;
```

The page number doesn't require the 32 bits, since pages are 4096 bytes long. The extra bits will be used to encode which cache file this page resides in. The priority field is a number representing this page's priority for being kicked out of the cache. How exactly this field is used hasn't yet been determined. The checksum is a (fast) page checksum that can be used to validate the contents of this page. Note that it will be useful to have a slower, more effective checksum for development and a faster (but less thorough) checksum for deployment.

The page status is an enumeration for the page's locking status (these are described in more detail later:

```
typedef enum
{
     PS_INVALID,
     PS_CLEAN_UNLOCKED,
     PS_CLEAN_LOCKED,
     PS_DIRTY_UNLOCKED,
     PS_DIRTY_LOCKED,
     PS_IN_FLIGHT
} PageStatus;
```

Note that this describes the layout of the tables in memory; how these data structures are represented on disk is described later.

The EFSD file handle is a small integer passed between the EFSD and the ECM. This is used opaquely by the EFSD and is used as an index into an open file table by the ECM.

```
typedef uint32 EFSDFileHandle;
```

The ECM request type specifies the request type to the rest of the system. Note that some "requests" are used to inform the prefetcher about the events handled solely by the ECM, and do not actually request that any particular action be taken by the prefetcher.

```
typedef enum
{
     ERT_READ,
     ERT_WRITE,
     ERT_READ_HIT,
```

```
    ERT_WRITE_HIT
} ECMRequestType;
```

The ECM request is a request descriptor that is used in various lists within the cache manager. These lists are doubly-linked, circular lists.

```
typedef struct _ECMRequest
{
    uint32 RequestID; /* same as EFSD request id */
    ECMRequestType RequestType;
    union {} Parameters; /* union of all parameters*/
    struct _ECMRequest *next;
    struct _ECMRequest *prev;
} ECMRequest;
```

The cache manager must maintain an array of files that have currently been opened by the EFSD. This array will be statically allocated. This will put a limit on the number of files that may be opened concurrently on the eStream file system. The elements of the array are the following:

```
typedef struct
{
    uint32 Valid;
    fileId File;
    HANDLE OpenFile; /* for simple implementation */
    eStreamInode *Inode; /* for robust implementation */
} OpenFileInfo;
```

The cache manager maintains a hash table containing information about each application that currently has open files. The hash table is indexed by app ID, and contains the following active app information records:

```
typedef struct
{
    AppID App; /* identity of this app */
    uint32 OpenFiles; /* # of open files */
    uint32 HaveAccessToken; /* boolean */
} ActiveAppInfo;
```

The ECM will use this table to quickly determine whether it should continue processing a request it gets from the EFSD, or if the request should be passed to the LSM to ensure that an access token is available. See the section below on ECM-LSM interaction for more details.

The LSM uses the access token state to specify a state for an access token. Right now, we only plan to support valid and invalid, but it may be interesting in the future to allow already opened files to be read, but no new files to be opened.

```
typedef enum
{
     ATS_INVALID,
     ATS_VALID,
     ATS_VALID_NO_OPEN
} AppTokenState;
```

Interface definitions

The ECM exports the following interfaces for operating on the cache. They may be called by the cache manager, prefetcher, or networking component. (Not all components are expected to call all interfaces; see each interface description for more details.)

Note that the cache interfaces are defined at a very high level as the actions that may be performed on the cache by the components, such as enqueuing a new request. They have been defined this way so that these intrinsic operations can be implemented correctly once and limit the possibility that an individual component will not perform proper actions.

ECMReservePage

eStreamStatus ECMReservePage(
    IN fileId *File*,
    IN EStreamPageNumber *Page,*
    IN ECMRequest **Request*
);

ECMReservePage reserves a page in the cache for a request. This interface is called by the prefetching component, and will send a request to the network component. Logically, this interface reserves an empty cache page for this request (if one is available), puts this request on the "in flight" queue, and calls on the network to request the page (unless it is already in flight.)

ECMIsPageInCache

    eStreamStatus ECMIsPageInCache(
    IN fileId *File*,
    IN EStreamPageNumber *Page*
);

ECMIsPageInCache returns TRUE if the specified block is in the cache, and FALSE otherwise. It is used by the EPF to determine if it should prefetch a block; normally, the EPF would choose not to prefetch something that is already in the cache. Note that it would be a good idea for the prefetcher to adjust the priority of a page that it thinks it wants to prefetch, so that they are less likely to be evicted from the cache before they are needed.

ECMDeplanePage

eStreamStatus ECMDeplanePage(

IN fileId *File,*
IN EStreamPageNumber *Page,*
IN char *Buffer[ESTREAM_PAGE_SIZE]*
);
ECMDeplanePage performs all the necessary actions for writing a page coming off the network into the cache and back to the EFSD. This consists of copying the page into the cache, remove all pending requests for this page from the in flight list, marking the page as clean/unlocked, and returning the page to the EFSD for each in flight request.

ECMReadPage

eStreamStatus ECMReadPage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);
ECMReadPage performs all the necessary actions for attempting a page read from the cache. The cache is checked to see if it contains the page; if so, the page is copied to the buffer, the EPF is notified of the hit, and appropriate status is returned. Otherwise, this page is put on the queue for requests pending to the prefetching component, and appropriate status is returned.

ECMWritePage

eStreamStatus ECMWritePage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);
ECMWritePage performs all the necessary actions for attempting to write a page in the cache. Note that this could be somewhat more complex than a read, because a partial write to a page might necessitate reading the page from the server before writing the partial page to the cache.

The following interfaces are the abstract interfaces that the ECM will use to communicate with the EFSD. Hiding the EFSD's raw DeviceIoControls behind these interfaces will help make porting the ECM into the kernel easier, should we decide to do that.

ECMSetTokenState

eStreamStatus ECMSetTokenState(
IN AppId *App,*
IN AppTokenState *State*
);
ECMSetTokenState is called by the LSM to indicate to the ECM that a token has become available or has expired. The main effect of this interface is to update the state of the specified application in the active app table. See the ECM-LSM interaction below for more details.

ECMGetCacheInfo

eStreamStatus ECMGetCacheInfo(
    OUT UNICODE_STRING *Location*,
    OUT uint32 **CurrentSize,*
    OUT uint32 **MaximumSize*
);

ECMGetCacheInfo is called by the client user interface to find out where the ECM cache is located and its current and maximum size. Location is an absolute path name of the cache file.

ECMSetCacheInfo

eStreamStatus ECMSetCacheInfo(
    IN UNICODE_STRING *Location*,
    IN uint32 *MaximumSize*
);

ECMSetCacheInfo is called by the user interface when a new cache location or size has been requested. Note that the cache manager may only begin using the new cache information after a restart of the client software (which may only occur on client machine reboot.) The client UI will call this interface when it wants to make a change; the ECM is responsible for actually resizing the cache and making any changes necessary to persistent storage (i.e. the registry).

EFSDGetRequest

eStreamStatus EFSDGetRequest(
    OUT EStreamRequest ***Request*
);

EFSDGetRequest reads the next request from the EFSD, including any parameters that need to be passed. This may involve one or more DeviceIoControl calls to the EFSD. **EFSDGetNextRequest** is responsible for allocating memory for this request, and an **EFSDCompleteRequest** call will be repsonsible for deallocating the memory.

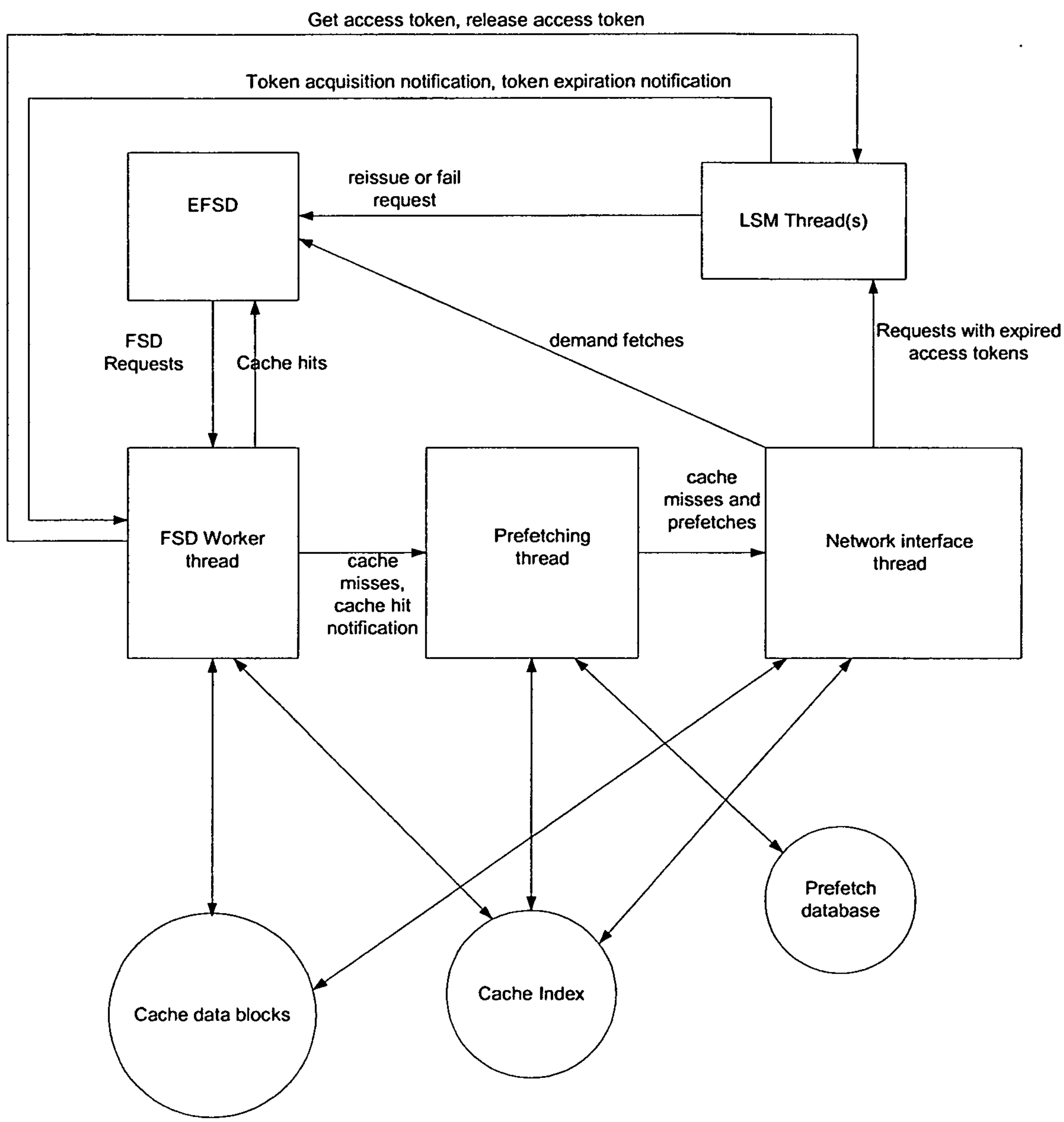
EFSDCompleteRequest

eStreamStatus EFSDCompleteRequest(
    IN EStreamRequest **Request,*
    IN ECMErrorCode *Status*
);

EFSDCompleteRequest will be called for each request that is received by the ECM via EFSDGetRequest. *status* indicates the completion status for this request, and may indicate success, a retry, or a particular failure condition. Non-persistent errors will be handled by the ECM internally or by requesting a retry of a particular request. Errors reported to the EFSD will be propagated up the file system stack.

Overall Client Architecture

The eStream client will have various types of threads in order to perform its work. The basic architecture is illustrated by the following diagram.



The FSD worker thread will pull requests from the FSD. It will return data for requests that can be satisfied immediately. Any request that requires information that is not currently in the cache will be put on a queue for the prefetching thread to handle.

The profiler will receive all cache misses from the FSD worker thread. Using its own data structures (which may include information about recent cache misses in addition to information about general prefetch patterns), it will decide which blocks it should prefetch. Demand fetch and prefetch requests are sent to the network component. The

only way demand fetches and prefetches are treated differently by the network component is that demand fetches are sent to the EFSD while prefetches are not.

The network thread will manage open connections to app servers and retry requests that time out. When data comes back from the network, the network thread will copy the returned buffer into the cache and to the FSD, if the request was a demand miss.

The cache manager consists of the EFSD worker thread and the APIs to access the cache index, the data blocks, and various queues used by threads in the client.

Not shown on the diagram is an error thread. This thread is responsible for calling the client UI module indicating appropriate error messages and waiting for the user's input. When any component decides that it has an error condition that requires user input, it calls **ECMReportError** with the request and an appropriate error condition, which will be enqueued for the error thread to handle. For example, when the network interface times out reading a page from an application server enough times, it will call **ECMReportError**. When the error thread gets to this request in the queue, it will ask the user if he wants to wait until the app server is available or allow the application to terminate.

ECM-LSM Interaction

The ECM-LSM interaction is a relatively simple one. The LSM notifies the ECM when it first receives an access token and when its access token expires. It does this via the **ECMSetTokenState** interface. The ECM keeps track of each application that has had files open, and whether or not we have an access token for each of these apps.

App ID	# of open files	Have access token?

Note that the LSM need not notify the ECM of mundane events like renewals as long as some token is valid. Also, the ECM does not keep track of the token itself, just whether or not we have a valid one. An additional nicety of this approach is that we could allow the ECM to satisfy requests out of the cache as if we have an access token, without actually having one.

When it receives a request, the ECM checks its table to determine if an access token is available. If it is, it handles the request as normal. If not, it asks the LSM to acquire an access token via **LSMGetAccessToken**. The LSM may return that it has a token, in which case the ECM will continue to process the request, or the LSM may say it doesn't have a token, in which case the LSM takes ownership of the request and will reissue the request when the access token is available.

When the number of open files drops from 1 to 0, the ECM will mark the token as invalid in its table and call **LSMReleaseToken**. The LSM may choose not to renew access tokens that have been released.

Component design

Two cache organizations will be presented. One is suitable for a quick implementation but doesn't lend itself particularly well to high performance or easy manageability; the other will be more difficult to implement but should provide better performance. I will first describe some data structures that are shared by both designs, then go into the specifics of each design.

Common Data Structures and Algorithms

Certain request lists are common to both cache organizations. One is a queue between the FSD worker thread and the prefetching thread for demand fetches that have not yet been seen by the prefetcher. The other is a list of all requests for pages that are "in flight." Requests from the in flight list are removed when they have been satisfied. The in flight list is unsorted and searched whenever a request comes back for requests that match the returned page. If the performance of this data structure becomes an issue, we will change its organization for faster lookup.

Both request lists use the request data structure described above.

The ECM will maintain an array of files currently opened by the EFSD. On file opens, an empty location in this table will be allocated for the newly opened file, and the index to that entry returned as the file handle. (Note that the way the interface between the ECM and the EFSD is defined, it is an error to open an already opened file. The cache manager will have to detect such cases and report an error, but it will not keep a reference count of the number of opens on each file.) This mechanism will allow the ECM to keep track of the volumes that currently have opened files as well as abstracting the client/server file ids away from the kernel driver. (This might allow us to update the client/server protocol without rewriting the EFSD.)

Easier Implementation

The cache will be implemented as a directory tree on the user's hard drive that parallels the eStream file system. Each file will contain a header and an array of status bytes in addition to the data blocks that the file contains. The array of status bytes has one byte for each page in the file. Each byte indicates the current status of that page in the file. (Pages have several different states, so a simple bit per page is not sufficient.) Each file will thus look like

Header
Page Status Bytes
File contents page 0

File contents page 1
...

The header is defined as:

```
typedef struct
{
uint32 magicCookie;
uint32 headerLength; /* Length of this header, in bytes */
fileId fileid; /* for sanity checking */
uint32 length; /* Length of the file, in bytes */
uint32 firstPage; /* Offset to the first page in the file
*/
Metadata metadata;
} ECMCacheFileHeader;
```

The page status bytes begin immediately following the header, and this area is padded with zeros to a page boundary. The first page of the file's contents (and thus each following page of file contents) will therefore begin on a page boundary.

Note that one issue with this design is that files that approach the file size limit of the underlying file system cannot be represented, due to the overhead with the header and bitmap. If this design is used solely for early engineering efforts, then this limitation is acceptable. If we have to work around this limitation, one way to do it is to make the headers and page status bytes reside in a separate file or files.

Directory contents would reside in server format in a file named "Directory" inside of the directory whose contents they represent (with the addition of the header and status bytes as described above for ordinary files). For example, z:\Program Files\Microsoft Office would reside in c:\Cache\Program Files\Microsoft Office\Directory. This has the drawback of creating special file names that can't be used by files in the eStream volume, but again, for an early engineering implementation, this is an acceptable limitation.

Another issue with deploying this implementation is that it is trivial to reverse-engineer this file format and copy files directly from the cache.

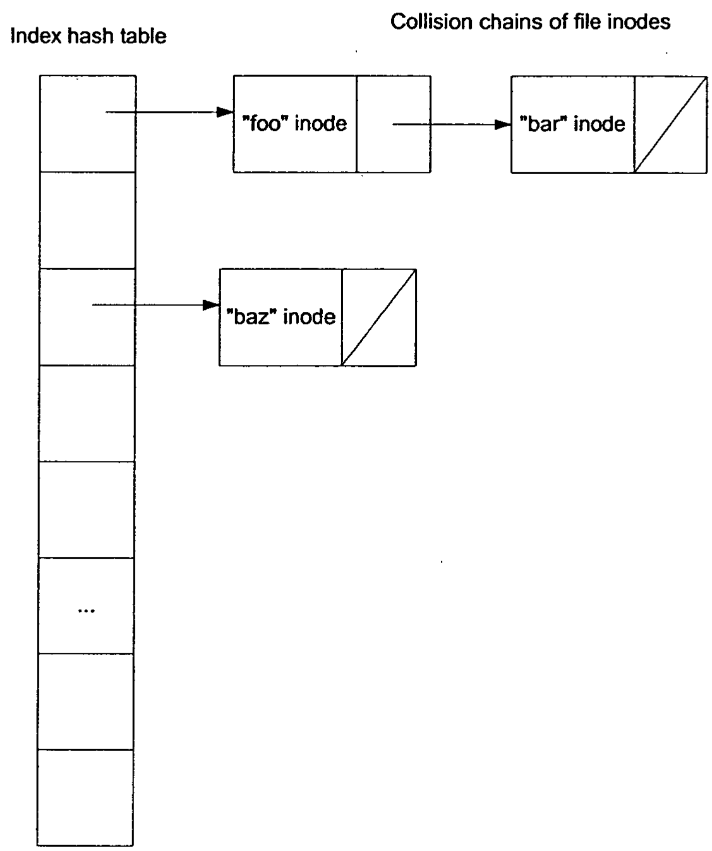
Robust Implementation

The cache will be organized into an index file and one or more cache data files. Multiple data files may be necessary as we may wish to allow the cache to grow larger than the 2 GB file size limit (for some native file systems) or to span multiple drive letters on the client. The data files will only contain pages of file content. These pages will be aligned on page boundaries. The index file contains all the information needed to locate file pages, and is contained in a separate file for simplicity.

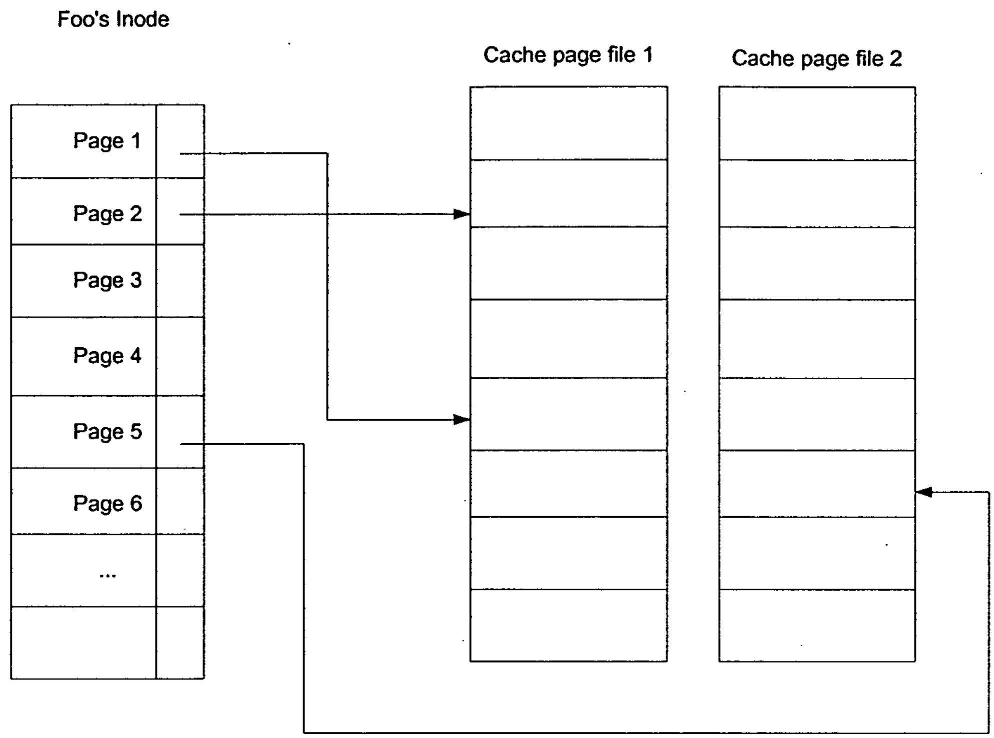
Page and index files must reside on a local disk (rather than a network disk) and cannot be shared by multiple clients.

Each file with any pages currently resident in cache will have a data structure containing information about that file, including its file id, the file id of the directory containing it, the file's metadata, and the map for finding the file's data blocks. This data structure is very similar to the inode of a traditional file system, and will be referred to as the eStream inode. A naive implementation of the inode is described above; no doubt, we will want to reorganize this data structure for more compact representation and better performance. Note that one requirement of the inode is that it contain a status field for each page in the file. One character is sufficient for this status; whether or not we can make do with fewer than 8 bits is an open question.

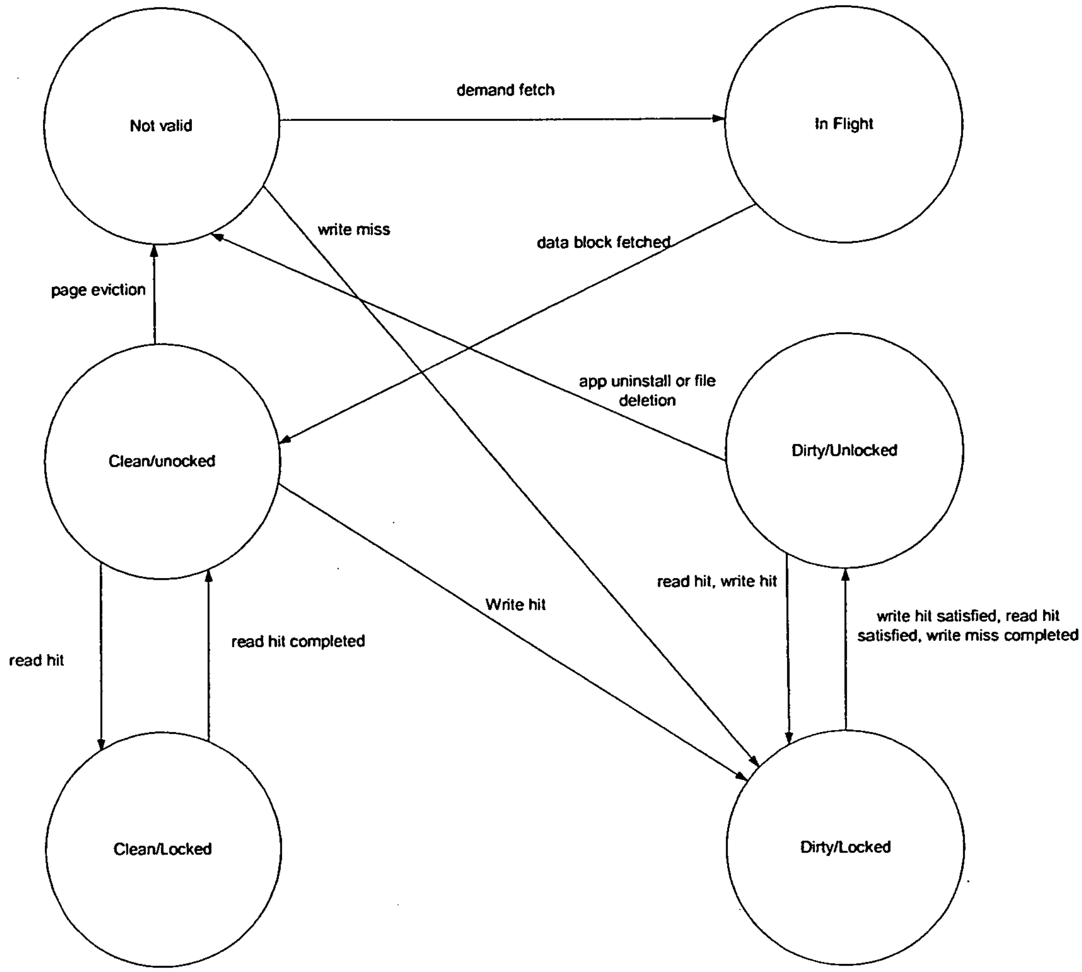
A hash table will be used to map file IDs to file inodes.



The inode contains pointers to each block's location in one or more cache page files:



To prevent race conditions, a single lock controls access to both the hash index and the linked list of requests that are pending network access. Individual pages in the cache may be locked for read or write access. Since each page's status is in the index, the index must be locked order to lock a page for reading or writing. The page states are controlled by the following state machine:



The dirty/clean distinction is between those pages that we have written locally (and thus cannot evict from the cache) and those pages that we haven't written (and thus can be refetched from the server).

A page would be locked while it was being read or written for copying to the file system driver. The operation may thus proceed with the index unlocked, without the possibility of page eviction while a copy is still in progress. The FSD worker thread is the only thread that reads or writes pages from the cache, so it's the only thread that can lock or unlock these pages. The in flight state is only for pages that are currently being fetched, either as a demand fetch or as a prefetch. The prefetching thread is the only thread that will put pages into this state, and only the networking thread will move pages from in flight to unlocked.

A list will be maintained of all "in-flight" requests. A single lock will control access to both this list and the cache index, so there are no race conditions between items being put on this list and data coming off the network. When the FSD worker thread gets a request, it acquires the index lock and looks at the status of the page. If the page is clean or dirty but unlocked, it will lock the page and copy it to the FSD. If the page is invalid, then this is a demand fetch, and the request is forwarded to the prefetcher. If the page is marked in

flight, then this is either a second request for an outstanding demand fetch, or it is a request for an in flight page. Either way, while this thread still holds the index lock, this request will be inserted into the list of in-flight requests. Race conditions might occur because the FSD might make multiple demand reads of the same page, or it may make a demand read to a page that is already in flight due to a prefetch.

Reading requested pages off the network and writing them to the cache (and to the file system driver, if necessary), are where this race condition comes up. We need to ensure that a request for a page that has arrived does not end up in the list of "in flight" requests. The solution is the following: When a data page comes back from the server, the networking component acquires the index lock to find the cache location of this incoming page. If the page is not marked in flight in the cache, this is a bug. (Of course, this is a relatively benign bug, and the NW component could just ignore the page.) The networking thread leaves the page as marked in flight, however, and unlocks the index. It writes the incoming page into the proper location, but it saves the in-memory copy of the page. It then reacquires the index lock, marks the page as clean/unlocked (since it's now in its final location in the cache), removes each request in the in-flight list for this page, then releases the lock. (Any further requests for the same page will find the page clean/unlocked, so the FSD worker thread will be able to satisfy these requests directly.) The networking component then proceeds to satisfy all of the requests it pulled off the in-flight list by using the copy of the page that it saved in memory. This way, it doesn't have to lock the index the entire time it is sending completed requests to the FSD.

Each of these complex scenarios is captured in the cache file's API's. As long as these are implemented correctly, other components don't need to worry about the exact sequence of operations that needs to occur.

Free Space Management
Free pages will be maintained as a free list in memory and as a bitmap on disk. The free list will be built from the bitmap on eStream client software startup. Access to the free list will be controlled by the same lock controlling access to the index.

Evicting Cache Pages
Individual cache pages may be evicted. There is an 8-bit field in the index for each page's importance. Initially, we will implement a random page replacement policy. Later, we will use this page importance field in an unspecified way to replace pages in such a way as to maximize interactive user performance and minimize application server load. Only clean/unlocked pages may be evicted. Pages that are evicted will eventually be put on the free list. Page eviction will only happen at "garbage collection" time. See "crash resilience and garbage collection," below.

Handling Cache Size
Growing the cache should not be an issue. The cache manipulation routines must know the overall size of the cache, in pages. Increasing the size of the cache on the fly should be a relatively straightforward process, as we merely need to lengthen the cache file(s) and add the new pages to the free page list.

Unfortunately, shrinking the cache is a much more difficult operation, since it potentially involves moving around pages that might currently be in use for paging operations or be in flight from the network. Changing the cache around at runtime is both difficult to implement correctly and a performance problem. The current plan is to support shrinking the cache only at eStream client software startup. The maximum allowed size of the cache will be stored in the Registry. On eStream client software startup, the current size of the cache will be compared against the allowed size specified in the registry; if it is larger than the maximum size specified in the registry, then the size of the cache will be reduced by evicting files and compacting the freed space. A request by the user to reduce the size of the cache will take effect the next time the client software starts.

Note that files that the user writes to the z: drive are not considered candidates for eviction (unless the file is explicitly deleted.) This means that the user's on-disk cache may in fact grow to be larger than the limit they specify.

Also note that at least one free page (not used by user-written files) is required for the file system to make forward progress. We also may want to require some minimal amount of cache before eStream will even run. Thus the maximum cache size specified by the user should be considered a "soft limit." There would be a "hard" minimum amount of space equal to the number of pages required to store the files written by the user on the z: drive plus a small amount of cache we designate just for running eStream. If this hard minimum is greater than the soft maximum specified by the user, the hard minimum would win. I would recommend preallocating and non-zero filling the file on disk so that we know that the space is available.

Crash Resilience and Garbage Collection

In order to provide crash resilience, the index will be periodically checkpointed to disk. Note that allocating blocks does not cause problems if the index is not updated. However, we cannot reuse a page's storage until that page has been marked free on disk.

The solution to this problem is to periodically garbage-collect the cache (if it is nearly full), and writing the index to disk. The cache manager will alternate between writing two cache index files. The index file will have a marker at the end that indicates that it has been successfully written and a time stamp, and on startup the ECM will use the latest, fully written index.

Data blocks will always be written directly to the cache page files. These files must be flushed before writing the index.

Garbage collection involves the following steps:
- lock the index
- copy the free list
- choose blocks in the cache to free, and make a list containing just the newly freed blocks. Mark these blocks as invalid in the file's inodes, but don't put them on the free list (yet)

- make a copy of the index
- unlock the index
- merge the list of newly freed blocks with the copy of the free list
- flush all cache page files
- write the new, merged free list (as a bitmap) and index to disk
- lock the index
- add the newly freed blocks to the free list
- unlock the index
- free any allocated data structures

Index File Contents
The index file contains the following items:
- List of cache block files, with their sizes
- Free block bitmap, per cache block file
- Inodes for all files; may be stored hashed or may be rehashed on startup.

Testing design

Unit testing plans

Cache file manipulation routines can be tested in isolation. We will write a standalone harness that exercises the functionality of the cache file manipulation routines by performing cache level operations directly. A multithreaded unit test for the cache manipulation routines would be ideal, so we can test the correctness and performance of our locking strategy without the need to build the entire cache manager.

Each "thread" of execution described by this document can be separately tested by creating a testing harness providing that thread inputs and monitoring its outputs. Replacements for the EFSD interfaces can be very effective here.

Stress testing plans

An interesting stress test for the cache manager is if it can work correctly with very small caches, even all the way down to 1 page. (Or at least, a cache with all pages but one marked as dirty.)

The cache manager will be able to operate in "verify mode," where requests that hit in the cache will still be sent to the server, and the pages returned by the server will be compared with the cached page's contents.

The cache manager will support multiple different page checksum algorithms. We can use a fast algorithm for deployment while using a more rigorous one in development. This also has the benefit of allowing us to test the performance impact of various checksum algorithms.

The cache manager will have the ability to verify the integrity of the cache index and free page bitmap. In particular, it will have the ability to determine taht no pages are allocated to more than one file in the file system, and that each page belongs to a file or is on the free list.

Stress testing for the ECM will include crash testing.

Cache manager testing will include resizing the cache.

Coverage testing plans

Cross-component testing plans

We can build a "cache only" file system by not using the prefetching and network components. This allows us to test the EFSD in conjunction with the cache manager without involving the prefetcher or the network component.

Early implementation of the client will likely involve a null prefetcher that does no prefetching.

We can use the testing harness for the cache manager that doesn't use the EFSD to drive the cache manager in conjunction with the prefetcher and network component. This allows us to test the combination of these components without driving it with the live file system driver.

Upgrading/Supportability/Deployment design

The client user-mode software and device drivers are packaged separately. (I.e. the client executable and the drivers are separate files on the disk.) This leads to the possibility of a "partial" upgrade that results in inconsistent versions of the drivers and client user-mode software. The drivers should support an interface that returns the version number of the driver, or of the interfaces provided by the driver. This will help the client software to recognize situations where it should tell the user to reinstall the client software and not result in bad system behavior.

Most (all?) on-disk data files should have file headers containing at a minimum a magic cookie and the file format version number. This will help us with upgrades in the future.

Open Issues

We need to address what happens when a fetch is requested and no empty space can be found in the cache. The prefetcher should probably block until such time as space is

made available for this request. While operating with very small amounts of cache will obviously cause bad performance, it should not result in a deadlock.

eStream Cache Manager Straw Man Proposal

Version 0.2

[redacted]

Purpose
The purpose of this document is to serve as the basis for the design of the eStream Cache Manager. As a straw man, this document is meant to serve as the basis for discussion, and anything here is subject to change. Assuming there are no major concerns with this document, I will proceed with producing a low level design for the cache manager.

Requirements in Brief
Support > 2GB client cache, possibly across multiple drives
Provide some level of protection against piracy, via both the file system and the cache
Fast lookup for what is in the cache and where to find it
Support automatic and user-specified cache size policies

As far as cache size goes, I think that it is reasonable for eStream 1.0 for the cache to be limited to one disk partition and 2GB of space, but the design should allow for very large caches (spanning more than one file and possibly more than one drive letter.) Note that if the cache is greater than 2GB in size, it cannot be mapped into the address space of a single process under NT/2000 on x86.

Cache Organization
The cache will be contained in 2 or more files. One file will contain the cache indices, and one or more files will contain the data blocks for cached files. (More than one cache data file may be required if the cache is larger than the largest file allowed on the native file system.) This allows us to keep the cache index file memory mapped and only map the data file(s) if there is enough memory space to do so.

Data Blocks
The cache data file will contain data pages frome the file system 4k in size.

Data will be stored in the cache uncompressed to allow easy page retrieval.

Cache Index
The cache index will be a b-tree. The key for the lookup will be the file id and page number requested. Keys in the b-tree are the set { volume #, file #, starting page, # of pages }. A lookup will succeed when the volume number and file number match, and the requested page is in the range from starting page to starting page + # of pages. The data stored for that key will be the offset into the cache for the beginning of the run. As is described in the file system proposal, the file number and starting pages are each 32 bits long. I propose making the starting page a 48 bit number and the number of pages a 16 bit number. This allows us to have a very large total cache and reasonable sized runs of contiguous pages in the cache.

Free space in the cache will have to be managed. Free blocks can be placed into a specially identified "free space file" in the index. Some auxiliary data structures may be convenient to make searching for a region of free space of a particular size.

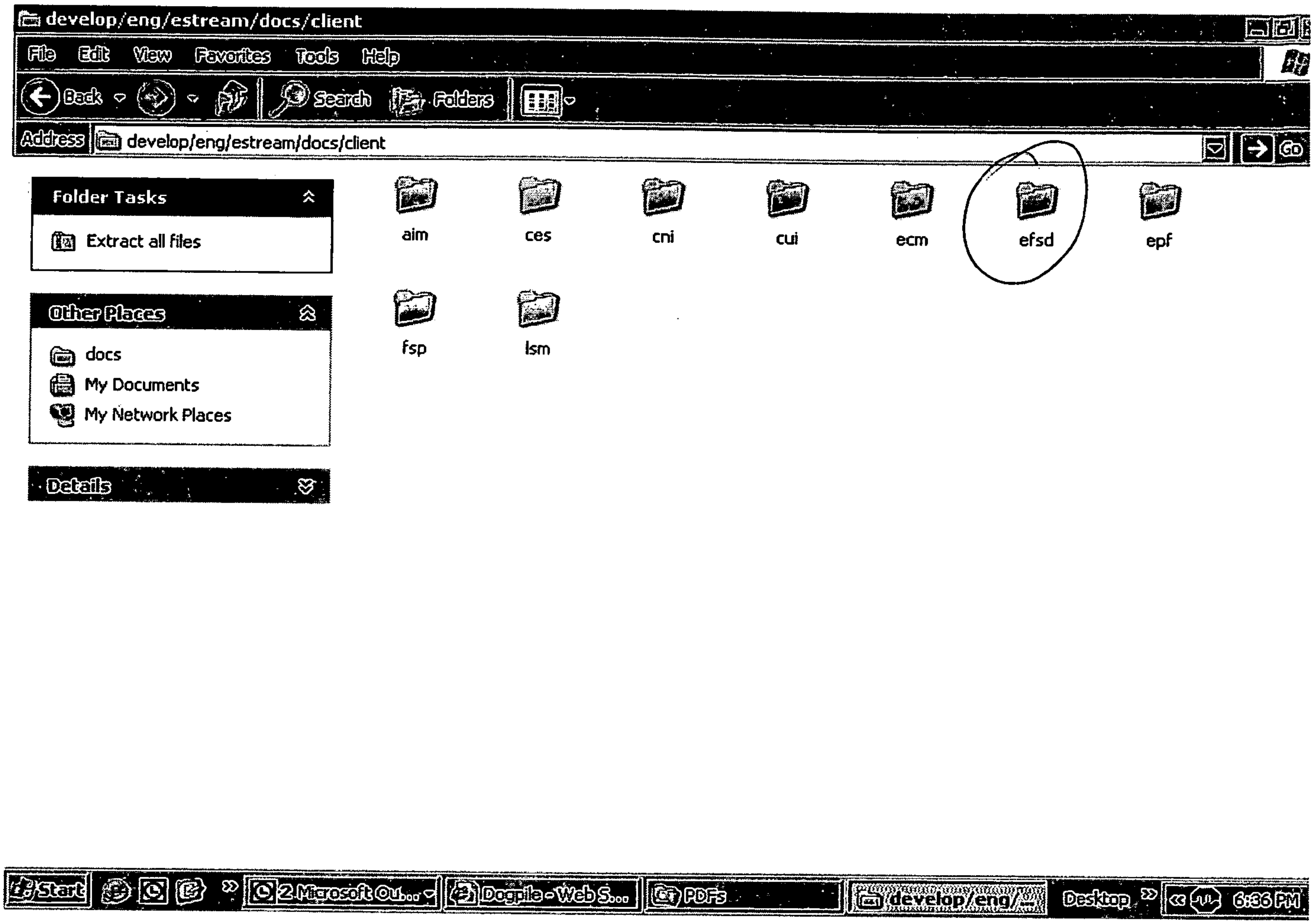
Metadata for a file would be stored in the cache. It would be indexed by page number -1 in the index.

Cache Replacement Policy
For simplicity, I propose that the cache manager evict entire files from the cache when it decides that it needs to clear room in the cache. (Of course, any fragmentary file that is in the cache can be evicted.) We should implement LRU for cache replacement, so we will evict files for apps that have not been run recently.

One Cache Per System
Administrator priviledges are required to install eStream. While various users on a system might have conflicting desires about eStream configuration, such as the size of the cache, I think that it is reasonable to have a policy where the adminstrator controls the setup of the eStream client. By limiting the cache to one per system, we eliminate any ambiguity about cache use in a multiuser environment.

Profiling and Prefetching
Profiling and prefetching have been broken out as a separate component in the client. It will be described elsewhere. It is expected that while the profiler/prefetcher will want access to the cache data structures (i.e. it wants to know what's already in the cache), the logic associated with prefetching is not logically tied to the cache manager, and should thus be separated.

Future Directions
Compression of the cache could potentially be a big win. We could provide cache compression similar to the way that NTFS provides file compression - we compress some number of blocks at a time (e.g. 16) and only store the compressed data when it saves at least one block of storage. Caching of data on disk can sometimes be a performance win, since decompressing the data can be faster than trasferring it on disk if the disk is slow enough.



[/STANDARD.01] [redacted]

eStream File System Driver Low Level Design

version 1.4
Curt Wohlgemuth
[redacted]

Functionality

The eStream Windows NT/2000 File System Driver (EFSD) is a kernel-mode file system driver to which file requests will be forwarded by the NT I/O Manager. It is the point of contact for users to access files on an eStream server. It works with the NT File Cache Manager to insure that kernel file caching is available for eStreamed files.

The Windows 98 EFSD is almost certainly to be very different from the driver for WNT and Win2K, and will not be described here.

In this document, I'm assuming that the EFSD communicates closely with the eStream cache manager (ECM) to perform the various file system requests. There may in fact be several components—if for example the ECM is broken into sub-components. Also, this document assumes that the ECM is in user mode; if this ends up in kernel mode, we will need significant changes to the interfaces to it.

Data type definitions

File handle

A file handle passed between the EFSD and the ECM is defined by the ECM:

```
typedef uint32 EFSDFileHandle;
```

Names

All file and directory names will be passed as counted Unicode strings, basically as defined by the NT header files. Note, however, that in NT the Buffer field is a pointer; for our purposes in communicating with the ECM, it's a NULL-terminated variable length array:

```
typedef struct _UNICODE_STRING {
   USHORT Length;
   USHORT MaximumLength;
   USHORT Buffer[1]; // NULL-terminated, 2-byte
                     // characters
} UNICODE_STRING;
```

Time stamps

The NT standard time format is a signed 8-byte integer representing the number of 100-nanosecond intervals since [redacted] These time stamps will be tracked for files and directories:

- Creation time
- Modification time

File attributes

File attributes are contained in an unsigned 4-byte integer. This subset of attributes from Windows NT will be supported:

```
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_TEMPORARY
```

These attributes are not supported:

```
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_DEVICE
FILE_ATTRIBUTE_SPARSE_FILE
FILE_ATTRIBUTE_REPARSE_POINT
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
FILE_ATTRIBUTE_ENCRYPTED
```

File size

File size will be represented as a 4-byte unsigned integer. Since sparse files are not supported, there will only be one file size passed between the ECM and the EFSD.

Metadata

This structure is defined to pass file and directory metadata between the EFSD and the ECM:

```
typedef struct { // 24 bytes, 4-byte aligned
   int64 CreateTime;
   int64 ModifyTime;
   uint32 FileSize;
   uint32 Attributes;
```

```
} Metadata;
```

Interface definitions

The EFSD is called by several different components, including

- the NT Executive (I/O Manager, Virtual Memory Manager), for standard file system requests
- the ECM, for these same file system requests, and to invalidate cached pages for coherency
- the client start software, to start and stop the EFSD

The EFSD supports standard FSD interfaces to the NT Executive modules; not all possible interfaces are supported, because the eStream file system is relatively low-functionality (compared to NTFS, for example).

The following file system requests will be supported; the interfaces for them will not be shown here, as they can be found in the DDK documentation.

- Create IRPs, for both new and existing files
- Cleanup and Close IRPs
- Read and Write IRPs:
  - synchronous and asynchronous
  - cached and non-cached
  - paging and non-paging
- Fast I/O reads and writes (with buffers or MDLs)
- File information (get and set) IRPs
- Directory query IRPs
- Volume information (get and set) IRPs
- File system information (get and set) IRPs
- Flush buffer IRPs
- System shutdown IRPs
- Various Fast I/O queries

The EFSD will not handle Directory Notification IRPs, nor will it support hard links (which are supported natively on NTFS on W2000 only): neither of these requests are required, and no expected user functionality will be lost without them. We are presently not supporting byte-locks; this may need to be revisited if the need arises.

In addition to the interfaces to the NT Executive, the EFSD will support various interfaces from other client components; all these will be sent via IOCTL calls. The first ones listed are simple support interfaces; the interfaces between the ECM and the EFSD follow these.

An IOCTL coming in to the kernel—via a DeviceIoControl() call—has the following parameters:

- IOCTL control code
- input buffer pointer
- input buffer size
- output buffer pointer
- output buffer size
- pointer to a 4-byte variable to receive the number of bytes written to the output buffer
- pointer to an OVERLAPPED structure for asynchronous operation (always should be NULL for EFSD)

All of the following interfaces are described in terms of the IOCTL buffers sent and received for each control code.

The following interfaces are called from the controlling client component (StartClient).

Starting and stopping the file system

The eStream FSD will be loaded into the kernel when a system is rebooted; i.e., it is always resident. If applications access files on this FSD via a drive letter, then the file system is implicitly turned off while a symbolic link for that drive letter is not present. Even when a drive letter symlink exists, the EFSD will not accept requests until the START IOCTL is sent.

These IOCTL control codes will be defined for starting and stopping the eStream FSD:

```
IOCTL_EFS_START_FS
IOCTL_EFS_STOP_FS
```

Starting the FSD

The input buffer for the START IOCTL should have the following:

- version id: 4-byte identifier for the client component
- debug flags: 4-byte value indicating the debug level to use

The output buffer for this IOCTL will be filled with the following:

- version id: 4-byte identifier for the EFSD version present
- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BAD_VERSION
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already started.

Stopping the FSD

The input buffer for the STOP IOCTL should have the following:

- force: 4-byte value
  - 0: shutdown only if no outstanding files are open
  - 1: shutdown regardless of state of open files

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already stopped.

Cache management interfaces

The following two interfaces are defined for use by the ECM to potentially invalidate data in the NT File Cache.

These IOCTL control codes will be defined for cache management for the eStream FSD:

```
IOCTL_EFS_INVALIDATE_FILE
IOCTL_EFS_INVALIDATE_DIR_CONTENTS
```

Invalidating a file

The input buffer for the INVALIDATE_FILE IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open file that must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

If in fact the file is open, but not present in the NT File Cache, this IOCTL will simply succeed; no error is returned.

Invalidating directory contents

The input buffer for the INVALIDATE_DIR_CONTENTS IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open directory whose contents must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

General file system requests

All file system requests that cannot be completely handled by the EFSD will be passed on to the ECM. Since the ECM is likely to be a user-mode service, the EFSD cannot call it directly; thus these "calls" are made by having the ECM send IOCTLs to the EFSD to get and fulfill requests. Each file system request requires multiple IOCTLs sent from the ECM to the EFSD:

1. The ECM sends an IOCTL to the EFSD to get the next request
2. The ECM sends a second and/or third IOCTL to finish the request

The following IOCTL control codes will be defined by the EFSD for use by the ECM:

```
IOCTL_EFS_GET_REQUEST
IOCTL_EFS_RETRY_REQUEST
IOCTL_EFS_GET_CREATE_NAME
IOCTL_EFS_FINISH_CREATE
IOCTL_EFS_FINISH_CLOSE
IOCTL_EFS_FINISH_READ
IOCTL_EFS_GET_WRITE_DATA
IOCTL_EFS_FINISH_WRITE
IOCTL_EFS_GET_RENAME_TARGET
IOCTL_EFS_FINISH_RENAME
IOCTL_EFS_FINISH_DELETE
IOCTL_EFS_FINISH_METADATA_READ
IOCTL_EFS_FINISH_METADATA_WRITE
```

For the DeviceIoControl() call sending IOCTL_EFS_GET_REQUEST, these parameters are invariant:

- the IO control code will be IOCTL_EFS_GET_REQUEST
- input buffer pointer will be NULL
- input buffer size will be 0
- output buffer must be non-NULL
- output buffer size must be **at least 40 bytes**—this is the largest buffer needed for any request (subject, of course, to slight modifications)
- pointer to bytes returned will be non-NULL
- overlapped pointer will be NULL

The IOCTL_EFS_RETRY_REQUEST is sent by the ECM (or some other user-space client component) to tell the EFSD that, yes, it needs to delete all intermediate information about a request already sent back with a GET_REQUEST call, and put the request back on the list for the ECM to retrieve. This eases implementation issues for the ECM. The input buffer for a RETRY_REQUEST is:

- request id of the previously retrieved request

There is no output buffer for a retry request IOCTL.

What follows is a list of file system requests from the NT I/O Manager, and the IOCTL calls needed from the ECM to service those requests. For all cases, if the EFSD writes to the output buffer for an IOCTL, the "bytes returned" field is written with the number of bytes written.

Create

This is used for both create and open, for files and directories.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Create request
- request id: a 4 byte field that will be subsequently sent in the calls to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- flags: 4 bytes, one or more of the following ORed together
    - CREATE_ONLY: fail if file exists already
    - OPEN_ONLY: fail if file does not exist already
    - TRUNCATE: overwrite existing file
    - DIRECTORY: create a directory
    - FILE: create a plain file
    - DELETE_ON_CLOSE: delete file on last close
    - IGNORE_CASE: obvious
- permissions: 4 bytes, one or more of the following ORed together
    - READ
    - WRITE
    - EXECUTE
- length of filename: 4 bytes, specifying the byte size needed for the Unicode string sent in the next call

Total size of output buffer: 24 bytes

GET_CREATE_NAME

The input buffer for this IOCTL should have the following data:

- request id: the id sent in the previous call

The output buffer for this IOCTL will be filled with the following information:

- request id for this transaction
- fully qualified name as a counted Unicode string (including drive letter, if any): the length needed was sent back in the GET_REQUEST call

FINISH_CREATE

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- handle: the 4-byte handle for this opened file, that can be used for subsequent file system requests. A unique value will indicate a bad handle, and a failed Create
- a Metadata buffer: the metadata for the created/opened file/directory.

The output buffer for this IOCTL should be NULL.

Note that a TRUNCATE Create request should cause the metadata sent back to reflect the possibly new (zero) length.

Close

This closes a handle of a previously opened file or directory. The EFSD will optionally send the updated metadata for this file in the GET_REQUEST output buffer. If the file has been modified in any way, the metadata fields will be non-zero; else they will all be zero.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Close request
- request id: 4 bytes, for use in subsequent calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for the previously opened file
- metadata for this file/directory: 24 bytes
    - creation time stamp
    - modification time stamp
    - file/directory size in bytes
    - attributes (as described above)

Total size of output buffer: 40 bytes

FINISH_CLOSE

The input buffer for this call should contain the following:

- request id for this transaction
- status: the NTSTATUS for this request

Read

This is used for reading file data.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Read request
- request id: a 4 byte field that will be subsequently sent in the IOCTL to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to read from
- length: 4 bytes, the length of the read, in bytes

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

Total size of output buffer: 24 bytes.

FINISH_READ

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- the number of bytes successfully read; 0 on error
- the data from the read; not present on error

The output buffer for this IOCTL should be NULL.

Write

This is used for writing file data.

GET_REQUEST

The output buffer for this will be filled with the following:

- type: a 4 byte field that indicates a Write request
- request id: a 4 byte field that will be subsequently sent in matching calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to write to
- length: 4 bytes, the length of the write, in bytes
- file length: 4 bytes, the length the file will be **if** this write succeeds

Total size of output buffer: 28 bytes.

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

GET_WRITE_DATA

This IOCTL will have an input buffer with:

- request id for this transaction
- status: if not STATUS_SUCCESS, this ends the request; the output buffer is untouched, and no FINISH_WRITE call is expected.

And the output buffer will be filled with:

- request id
- data buffer for the write—the byte length sent in the previous GET_REQUEST

FINISH_WRITE

For this finishing request IOCTL, the input buffer has these contents:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- bytes actually written; should be equal to requested bytes unless failure occurs.

Rename

This is used for renaming a file or directory.

GET_REQUEST

The output buffer for this IOCTL will be filled with the following:

- type: a 4 byte field that indicates a Rename request
- request id: a 4 byte field that will be subsequently sent in the FINISH_REQUEST call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- length of target name: 4 bytes; the byte length needed for a counted Unicode string for the target name

Total size of output buffer: 20 bytes.

GET_RENAME_TARGET

The input buffer for this call will have the following:

- request id for this transaction
- status: if not STATUS_SUCCESS, then this terminates the request: the output buffer is not touched, and no FINISH_RENAME call should be sent

The output buffer will be filled with the following:

- request id
- target name: a counted Unicode string, using the same number of bytes as sent in the GET_REQUEST output buffer

FINISH_RENAME

The input buffer for this call should have the following:

- request id
- status: NTSTATUS for the transaction

Delete

This is used for deleting a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Delete request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_DELETE

The output buffer for this call should be NULL.

The input buffer should have the following contents:

- request id: matching id from the GET_REQUEST call
- status: NTSTATUS of this request

Metadata read

This is used for requesting metadata about a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_METADATA_READ

The output buffer for this IOCTL will be NULL.

The input buffer should have the following contents:

- request id: id from the corresponding GET_REQUEST
- status: NTSTATUS for this operation
- the following data about the file or directory:
    - creation time stamp
    - modification time stamp
    - file/directory size in bytes
    - attributes (as described above)

Metadata write

This is used for setting metadata for a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata Write request
- request id: a 4 byte field that will be subsequently used for all calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- metadata for this file/directory: 24 bytes
    - creation time stamp
    - modification time stamp

- file/directory size in bytes
- attributes (as described above)

Total size of output buffer: 40 bytes.

FINISH_METADATA_WRITE

The input buffer should have the following contents:

- request id: from the previous call
- status: NTSTATUS for this request

Component design

This section is organized in the following manner:

1. General layout of the eStream file system driver
2. General observations about the low level design
3. Organization of data structures
4. Description of the algorithms for communication with the ECM
5. Description of each dispatch routine

Layout

The EFSD will be generally organized in the following manner:

- All major IRPs will have their own dispatch routine.
- All actual I/O requests to the ECM will be generalized from the dispatch routines to a set of routines that handle the communication with the ECM, to isolate this aspect.
- All utility functions will be in their own file or files.

General points

The design of the EFSD will look a lot like the sample FSD from Rajeev Nagar's NT FS Internals book, which looks a whole lot like the Fastfat FSD source from the NT IFS kit.

Here is a list of general points that can be made about the EFSD:

- Any IRP that can be handled asynchronously will be posted to a work queue; this means that the dispatch routine for such an IRP must be able to handle being called in a context other than the original requestor.
- There are no volumes, and no Volume Parameter Block or Volume Control Block. There isn't a VPB for a network redirector; I've verified this with the

LanManager redirector. Hence we don't have to support any operations on a volume in EFS.

- We will not allow the creation of paging files in EFS. There is a bit available for a Create IRP that specifies this, and we can complete the IRP with an unimplemented error return code.
- All file synchronization will be on a File Control Block (FCB) basis, using the standard Resource and PagingIoResource ERESOURCE objects used by the rest of the Windows Executive.
  - User requests will be synchronized by acquiring the main Resource—shared for reads, shared for most writes, and exclusive for file size changes, deletion, etc.
  - Paging I/O requests will be synchronized by acquiring the PagingIoResource—again, shared for reads, shared for most writes. Exclusive access will be needed to set file sizes.
- Most disk file systems have a resource associated with a VCB, which is acquired exclusively for creation/deletion etc. We will have a global EFS resource for this, since there are no VCBs.
- Asynchronous requests will be handled by posting the IRP to the CriticalWorkQueue, and marking the IRP as pending.
  - A common worker routine will be used for all async posts, which will dispatch the IRP to the appropriate real IRP routine when it's invoked.
  - An async request will be defined as one that IoIsOperationSynchronous() returns false, and the EFSD is the top-level component (see below)
- The EFSD will track the top-level IRP for the thread whose context it is running in. In particular,
  - No async processing request will be honored unless the EFSD is the top-level component
  - No cache manager requests will be made unless the EFSD is the top-level component
  - EndOfFile size—that is, the true size of the file—will not be extended or changed by paging I/O
- EFS will not support holes in files, and hence the ValidDataLength FCB field will be set to disable this. This means the AllocationSize for an eStream file/directory will always be equal to the EOF size.
- Most fast I/O routines will be supported in EFS. We will use the FSRTL supplied routines for fast reads and writes.
- All cache manager resource acquire/release callbacks will be supported. All will point to common routines that simply acquire or release the main or paging I/O resource for the FCB. The Context pointer passed into all of them will be the FCB for the stream.
- Synchronous read/write requests will update the CurrentByteOffset in the File object.
- Each Create will result in a unique Context Control Block (CCB) data structure; this will be small, and only hold those few fields needed:
  - For the Directory Control IRP, a CCB needs to hold the current entry index and the pattern originally used—for subsequent queries

    - A field for various flags
- A single FCB will represent all current open instances of a file. When a Create request comes in, the EFSD will search the current open FCBs to try to find one matching this file/directory name.
    - For now, this will be a hash table on the file name. We can improve this as needed.
    - The EFS global resource must be acquired exclusively:
        - before the global FCB data structure is searched. **Why? If it's just a read, can't we acquire it non-exclusively?**
        - before a new FCB is added to the list
        - before an FCB is deleted from the list
- EFS will not support open by file ID; hence the FileInternalInformation class for a File Information IRP will not be supported.
- Actual I/O will be directed to standard routines in a separate file, so they can be isolated and updated easily as our method of transferring data changes.
- Here's how to do file/directory renames:
    - The I/O manager will send to the EFSD this sequence:
        - Create for source
        - Create for target, with the SL_OPEN_TARGET_DIRECTORY flag set
        - Set Information with a Rename request for the source, sending the target directory FileObject handle and the target name in the FileInformationClass record.
    - EFSD needs to do this:
        - When it receives the Create for the target and the target *directory* exists, return STATUS_SUCCESS, and change the name in the FileObject to the basename of the target (the full pathname of the target is sent in), and set the Status.Information to FILE EXISTS or FILE DOES NOT EXIST, as appropriate. If the target directory doesn't even exist, return PATH NOT FOUND.
        - When it receives the Set Info request, if all the flags check out (e.g., if the file exists, ReplaceExisting must be TRUE), send a Rename request to the ECM.
- Reads and writes to only regular files will be supported, not to directories.
- Any code that touches user buffers or can call routines that may throw exceptions must be guarded by a try/except block.
- Some tips on memory allocation (from /perforce-doc-dir/osrdocs/defensive-driv.html)
    - We will use our own memory allocation/deallocation routines, instead of ExAllocatePool() et al. directly
    - These routines can do various checks for trashing memory:
        - fill allocated memory with a pre-defined bit pattern, instead of zeroes; fill deallocated memory with a different pattern.
        - allocate a header/trailer with standard information, like where allocated, from what pool, etc.

  - change the bit pattern in the header/trailer on deallocation, and look for freeing memory twice
- We probably want to allocate using lookaside lists, since we'll be allocating and deallocating smallish chunks of memory for our data structures.

Data structures

The following are major data structures used internally by the EFSD. Data structures used to communicate with the ECM are described in the following section.

NodeIdentifier

A NodeIdentifier is a simple structure that starts all other structures used in the EFSD. This makes a good debugging check to insure that we receive and are operating on the right type of data. It consists of two fields: an identifier field, and the total structure size.

FCB

The FCB is a critical data structure for the file system driver. There is one FCB structure allocated for each unique file or directory that is currently open—regardless of how many open handles there are for this entry. Multiple CCB structures can point to a single FCB.

Logically at least, part of an FCB is exposed to the Cache Manager and VMM to support caching and paging I/O. We will follow the example of Rajeev Nagar's book, and embed the FSRTL_COMMON_FCB_HEADER and other required structures directly in an FCB.

Here are the basic contents of an EFSD FCB:

- The required FCB contents above
- An open handle count: incremented on Create, decremented on Cleanup
- A reference count: incremented on Create, decremented on Close. The semantics of NT file requests require these two be used together to determine when an FCB can be deleted
- A pointer to the next FCB on its hash list
- A pointer to the first CCB opened for this FCB
- The fully qualified name of the file/directory
- The Metadata associated with this file/directory
- A SHARE_ACCESS structure used to check sharing violations
- Various flag bits

CCB

A CCB represents a currently opened handle to a file or directory. If two processes have the same file opened, there will be a unique CCB allocated for each process.

Here are the basic contents of an EFSD CCB:

- A pointer to its corresponding FCB
- A pointer to the next CCB opened for its FCB
- A pointer to the FileObject opened for this file/directory
- The current file index for a directory query
- The directory search pattern used for directory queries for this opened handle
- Various flag bits needed

IrpContext

An IrpContext structure encapsulates the interesting data from an IRP, and the current stack location, for easy access. This structure is allocated on entry to dispatch routines, and used during processing, before being deallocated on exit.

Communicating with the ECM

I tend to divide a file system driver into two logical parts:

1. A front-end that understands the NT FSD interfaces and semantics
2. A back-end that actually perform the requested actions

Our back-end is the (admittedly ugly) interface for communicating with the ECM, which currently sits in user-space. It's very important to design the ECM such that its front-end and back-end are nicely separated: since the ECM may move to kernel space, or we might find better interfaces for them to communicate with each other, we need to design with this in mind.

Given the current ECM interfaces defined above, here is a basic design to handle them:

- An EfsdRequest object will be created for each request that must be handled by the ECM.
- Each dispatch routine that results in a request to the ECM will allocate an EfsdRequest and send it to a common routine for further processing.
- New requests will be placed in a NewRequest queue.
- Requests that have been "sent" to the ECM, but not yet finished, will be removed from the NewRequest queue and placed in a PendingRequest list.
- Finishing a request entails removing it from the PendingRequest list, returning the contents to the dispatch routine, and destroying the request object.

Data structures

EfsdRequest

This contains:

- request id: a number that uniquely identifies this request
- type: the type of request (e.g., Create, Write)
- FCB pointer for the file/directory for this request
- IrpContext pointer for this request
- a kernel event object used for signaling that the request is satisfied

NewRequestSemaphore

The EFSD device object's device extension will contain a semaphore dispatcher object, initialized to non-signaled, and with an initial limit of MAX_LONG. When a request is added to the NewRequest queue, this semaphore is released (and the count is incremented by 1); the GET_REQUEST IOCTL will wait on this semaphore object (which decrements the count by 1).

NewRequestQueue

This is actually a kernel-managed interlocked list that is allocated in the device extension area, guarded by a spin lock that's also located in the device extension. Requests will be added to the tail, retrieved from the head.

PendingRequestList

This list must be searched when an ECM non-GET_REQUEST IOCTL is received, so we can't use an interlocked list. We'll use a global single-linked list structure, with elements allocated from a dedicated lookaside list using non-paged memory. Before a thread can access the list elements, it must acquire a mutex.

Algorithm

- All dispatch routines, and asynchronous read/write routines, will call LowLevel-PostRequest() to have their requests satisfied. LowLevelPostRequest() is itself synchronous; that is, the code calling it is something like this:

```
status = LowLevelPostRequest(fcb, irp_context);
     // we're done;
     // do all deallocation and cleanup needed
IoCompleteRequest(irp, ...);
```

- LowLevelPostRequest() will do the following:

```
if this is a read or write IrpContext
    see how many requests are needed to satisfy the IRP: can't span page boundary
    allocate the N requests
    allocate an array of N event pointers to hold the request events
    assign the pointers to the array
    if > THREAD_WAIT_OBJECTS
        allocate an array of N PKWAIT_BLOCKS
```

```
else if this is a directory query IrpContext
    look at the FileSize of the directory FCB
    allocate enough read requests to read all directory data from the ECM
    as above, allocate an array of event pointers, wait objects (if necessary)
else
    allocate a new EfsdRequest for the incoming FCB and IrpContext
place all requests on the NewRequestQueue, using ExInterlockedInsertTailList()
call KeReleaseSemaphore() on the NewRequestSemaphore
if multiple requests generated
    call KeWaitForMultipleObjects() on the request object's event
else
    do a KeWaitForSingleObject() on the request object's event
when the event(s) is/are signaled
    fill in the values into the IRP
    deallocate the EfsdRequest(s)
    return status
```

- When a GET_REQUEST IOCTL comes in from the ECM, the EFSD will do the following:

```
do a KeWaitForSingleObject() on the NewRequestSemaphore
remove the first request from the NewRequestQueue,
                using ExInterlockedRemoveHeadList()
lock the PendingRequestList mutex
place this request on this list
release the mutex
fill in the IOCTL output buffer identifying the request
complete the IOCTL IRP
```

- When a RETRY_REQUEST IOCTL is received, the following takes place:

```
lock the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
enqueue the request on the NewRequestQueue—on the list head
release the NewRequestSemaphore
complete the IOCTL IRP
```

- When any "finishing" IOCTL is received—i.e., the second of two or the third of three—the following is done:

```
acquire the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
do all buffer copying, set all flags,
            and otherwise insure the input IRP has the correct state
signal the EfsdRequest event
complete the IOCTL IRP
```

- When any other request IOCTL is received—e.g., the second of three—this is done:

```
        acquire the PendingRequestList mutex
        search the list by request id; error if not found
        release the mutex
        fill in the output buffer of the IOCTL as appropriate
        complete the IOCTL IRP
```

Dispatch routines

DriverEntry

This does a whole slew of initialization, including the dispatch table, fast I/O table, the cache callbacks, the FCB hash table and its synchronization object, creates the FS device object, and sets up the interface with the ECM.

Create

There is one Create routine; there will be no async processing of Create requests. Ultimately, its job is to send a create request to the ECM, and return SUCCESS or not to its caller. Here is a general algorithm for this routine.

```
create an IrpContext
if a page file is requested
    return error
generate the absolute pathname of the requested file:
    if OPEN_TARGET_DIRECTORY specified
        if OPEN_ONLY not specified
            return error
        generate the pathname of the parent directory of the requested file
    if a related file object is specified
        if the related file is not a directory
            return error
        if the related filename doesn't start with '/'
            return error
        if the input filename starts with '/'
            return error
        concatenate the input filename with the related file directory
    else use the input filename
        if the input filename does not start with '/'
            return error
acquire the global EFS resource exclusively
search the FCB hash table for this file by name
if not found
    call LowLevelPostRequest() to send the request to the ECM
    if error is returned from ECM
        return the correct error: FILE NOT FOUND or PATH NOT FOUND
    create a new FCB and add to the hash table
    call IoSetShareAccess() for this FCB
else
    check input attributes/flags against those in FCB
        if opening for write or delete on close
            call MmFlushImageSection()
            if this fails
```

```
                return error
        if failing mismatch—e.g., if IoCheckShareAccess() fails
            return error
    create a new CCB for this file
    set all appropriate flags on CCB and/or FCB
        COMMON_FCB_HEADER flag is set to FastIoIsPossible
    set all fields on input FileObject:
        write through flag
        FsContext points to common FCB header
        FsContext2 points to CCB
    if OPEN_TARGET_DIRECTORY is specified
        search for the input target object:
            look for this in the FCB hash table
            if not found
                send a Create request for this file to the ECM
        if this target filename exists
            set Status.Information to FILE_EXISTS
        else
            set Status.Information to FILE_DOES_NOT_EXIST
        if a Create was sent to the ECM for the input FileObject
            send a Close request for the input target to the ECM
        change the name in the FileObject to the basename of this file
        the CCB and FCB remain opened for the target directory
    all necessary data structures should be deallocated, for success and error
    release the global EFS resource
```

Cleanup

No async posting of Cleanup requests will be done. Algorithm:

```
    acquire the global EFS resource, and the FCB Resource, for this file, exclusively
    if this file is marked for deletion
        if this is the last open handle for the file
            acquire the PagingIoResource exclusively
            set the file size in the FCB to 0
            release the PagingIoResource
            purge the cache, if necessary, with MmFlushImageSection()
            call LowLevelPostRequest() to send a Delete request to the ECM
    decrement the count of open handles in the FCB
    if caching is on
        flush the cache by calling CcUnitializeCacheMaps()
    any time stamps must be updated if accesses were done using fast I/O
    set the FO_CLEANUP_COMPLETE flag in the FileObject
    call IoRemoveShareAccess()
    release the global EFS and FCB Resources
```

Close

There will be no async posting of Close requests. Note that we only send a Close request to the ECM if this is the last close for an open file—i.e., we're matching Close requests with Create requests.

Algorithm:

```
acquire the global EFS resource exclusively and the FCB Resource
deallocate the CCB
decrement the reference count for the FCB
if the FCB ref count is now 0
    remove the FCB from the hash table
    deallocate the FCB
    call LowLevelPostRequest() to send a Close request to the ECM,
                                          updating metadata if necessary
release the global EFS resource and the FCB Resource
```

Read

Reads will definitely be open to async posting. A general algorithm:

```
if the read length is 0
    return success
if the target file object is a directory
    return error

if this IRP can be handled async
    post for async processing

if this read is non-buffered, and it's not for paging I/O, and
                        there is a mapped data section object for this file
    acquire the FCB main Resource exclusive, and the PagingIoResource shared
    call CcCacheFlush() on the range of this read (current byte offset, length)
    release the FCB resources
if this is for paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared
if this read starts beyond EOF
    return EOF
if the read length goes beyond EOF
    truncate the length
if this is a buffered read
    if the PrivateCacheMap is NULL
        call CcInitializeCacheMap()
    if this is an MDL read
        call CcReadMdl()
    else
        call CcCopyRead(), using either an allocated MDL or the UserBuffer

else (it's non-buffered)
    call LowLevelPostRequest() to send a Read request to the ECM

if this is a synchronous, non-paging read
    update the current byte offset in the FileObject
set the number of bytes read in the Status.Information field
release any acquired resource and deallocate appropriate data structures
```

Write

Writes will definitely be open to async posting. A general algorithm:

```
if the write length is 0
    return success
if the input file is a directory
    return error
if the file not opened with write permissions
    return error

if this IRP can be processed asynchronously
    post for async processing

if this is a buffered write
    call CcCanIWrite()
    if false
        we have a hard error; fail
if this is paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared
if the length is
        (Low == FILE_WRITE_TO_END_OF_FILE) && (High == 0xffffffff)
    we're to write at EOF
if this is a non-paging, non-buffered write, and
                        there is a mapped data section object for this file
    acquire the global EFS resource exclusive
    acquire the PagingIoResource shared, starving exclusive waiters
    call CcCacheFlush() on the range for this write
    release the PagingIoResource
    return error if the cache flush failed
    acquire and release the PagingIoResource exclusive (to serialize)
    call CcPurgeCacheSection() on the range for this write
    release the global EFS resource

if this is a paging I/O write
    if the starting offset is beyond the current EOF
        return success
    if the ending offset is beyond the current EOF
        truncate the write length to EOF

if this is a buffered write
    if the private cache map is NULL
        call CcInitializeCacheMap() for this file
    if the write will extend the file size
        release the resource acquired
        re-acquire the resource exclusive
        call CcSetFileSizes() inform the cache manager
    if this is an MDL write
        call CcPrepareMdlWrite()
    else
        call CcCopyWrite(), using either an allocated MDL or the UserBuffer

else
    call LowLevelPostRequest() to send the write request to the ECM

set the number of bytes written in the Status.Information field
update the file size fields in the FCB if this write extends the length
if this is a non-paging write
```

```
        set the CCB modification flag
        if this write is synchronous
            update the CurrentByteOffset field in the FileObject
    release any acquired resource and deallocate appropriate data structures
```

Fast I/O Read

Initially at least, we'll just set the fast I/O read routine to FsRtlCopyRead().

Fast I/O Write

Initially at least, we'll just set the fast I/O write routine to FsRtlCopyWrite().

Fast I/O Query Basic Info

This will just fill in the basic info buffer with the data in the FCB.

Fast I/O Query Standard Info

This will just fill in the standard info buffer with the data in the FCB.

Fast I/O Query Open

This will just fill in the network open info buffer with the data in the FCB, if the file exists. Some empirical observations I've made using NTFS:

- Regardless of whether the file exists or not, this will return TRUE (all fast I/O routines are boolean)
- If the file does not exist, it will set the EOF size in the buffer to 0. The AllocationSize must be non-zero. All other fields seem to be don't cares.
- If the file exists but is zero length, then both EOF and AllocationSize will be 0.
- The IRP sent to this routine is for an IRP_MJ_CREATE; we can use more than just the name to identify the file, but also the security characteristics or whatever else is sent in the IRP.

File Query Info

Standard queries will be supported; these however will not:

- FileInternalInformation—no OPEN_BY_FILE_ID
- FileEaInformation—no EA data
- FileCompressionInformation—no on-disk compression
- FileStreamInformation—no multiple streams

File Set Info

These actions will be supported:

- EndOfFile size changes
- AllocationSize changes
- Time stamp changes
- File position changes
- File disposition changes—delete pending
- File rename requests

Here is a general algorithm:

```
if AllocationSize or EOF size change
    if the new size is smaller than the current size
        call MmCanFileBeTruncated(), to ask the VMM if this is okay
        if yes
            call CcSetFileSizes()
        else
            return STATUS_USER_MAPPED_FILE error
    else
        send a Metadata Set request to the ECM with new, extended size
        if status returned is error
            return error (disk space full)
    update AllocationSize and FileSize fields of required FCB header

else if time stamp change
    send Metadata Set request to ECM
    if error returned
        return with error

else if file position change
    update FileObject CurrentByteOffset field

else if disposition (delete) change
    if the Delete flag in the IRP not set
        clear delete on close FCB flag
    if file already marked for delete on close
        return success
    if file not open with write permission
        return error
    if MmFlushImageSection() fails
        return error
    if this is a directory, and the directory is not empty
        return error
    set delete on close flag in FCB

else if rename change
    if the source name is for a directory
        if the directory has any open files/directories under it
            return error

    if Parameters.SetFile.FileObject is NULL (simple rename)
        target dirname is the same as dirname of IRP FileObject
        target basename is Buffer.FileName

    else
        if Buffer.RootDirectory is NULL (fully-qualified rename)
```

```
            fully qualified target name is Buffer.FileName

        else  (relative rename)
            call ObReferenceObjectByHandle() to get the file object of the relative directory
            from file object, get (fully qualified) dirname of target
            append Buffer.Filename to root directory dirname to get fully qualified target

    if the target name isn't on EFS
        return error
    if target exists
        if Parameters.SetFile.ReplaceIfExists is FALSE
            return error
        if target is a directory
            return error
        if target is read-only
            return error
        if target has any open handles to it
            return error

    call LowLevelPostRequest() to send a Rename request to the ECM
```

Directory Query

Directory queries turn into directory read requests to the ECM. The EFSD will take the contents of the read buffers and fill the IRP_MN_QUERY_DIRECTORY buffers sent to it from the NT Executive by parsing the directory contents.

Note: this design does not use the NT Cache Manager for metadata or for directory entries, nor does the EFSD store the directory contents anywhere in its data structures. It will always go back to the ECM for directory queries. Given that most directory queries occur in this order:

Create
Directory query
Close

unless we can cache the contents in a location more persistent than an FCB, we will need to resubmit the request to the ECM for each new directory query. If this poses a performance problem, we need to handle it then.

Directory queries are subject to async processing.

This is an ugly NT interface. Here are some general points regarding directory queries, and how they will be implemented on EFS:

- These requests come in from the I/O Manager in a context-sensitive sequence. I.e., a request will come for the initial N directory entries; the next request will be for the next M entries; etc. Kind of like strtok().
- Thus, state must be maintained from request to request. This state will be kept in the CCB for a file stream, and consists of:

    - Pattern sent in on first request
    - Index of n'th entry to start retrieving with
- My experience is that the INDEX_SPECIFIED flag is **never** set in a directory control query, even on queries subsequent to the initial one.
- For EFSD, the FileIndex will represent the offset of the fixed-length portion of a directory entry as returned by the ECM.
- Initially, at least, **all** directory queries will cause the EFSD to read all the entries for that directory from the ECM. We can modify this later if needed.

Here is a general algorithm, based on the sources for the Fastfat driver:

```
if the FileObject is not for a directory
    return error

post this for async handling, to read all directory pages from the ECM

if the CCB pattern field is empty and the CCB flag "match all" isn't set
    this is the initial query
    acquire the FCB Resource exclusive
else
    acquire the FCB Resource shared

get a pointer to the input buffer, using either an MDL or the UserBuffer
if this is the initial query
    parse the FileName pattern
    save the pattern in the CCB
    if the pattern is "*"
        set the "match all" flag in the CCB

if SL_RESTART_SCAN is specified
    use an index of 0 for the query
else if SL_INDEX_SPECIFIED is specified
    use the input index for the query
else
    if this is the initial query
        use an index of 0
    else
        use the index saved in the CCB

start with the directory entry corresponding to the starting index
if this index is beyond the number of entries in the directory, and this is not the initial query
    return STATUS_NO_MORE_FILES
while there are more directory entries
    if the directory entry name matches the pattern in the CCB,
                                        using FsRtlIsNameInExpression()
        if there isn't room in the buffer for this entry
            break
        write the entry in the input buffer
        if there is a next directory entry beyond the current one
            the FileIndex field is set to the offset of this next directory entry
        else
            the FileIndex field is set to 0
        if there is a previously written entry
            fix up the NextEntryOffset in the previous entry to the byte offset of this entry
```

```
        if SL_RETURN_SINGLE_ENTRY specified
            break
        8-byte align the current pointer in the user buffer
    advance to next directory entry

if we wrote nothing
    if we stopped because of no room
        return STATUS_BUFFER_OVERFLOW
    else
        return STATUS_NO_SUCH_FILE

update the index field in the CCB
Status.Information is set to the number of bytes written
return STATUS_SUCCESS
```

File System Query Info

Empirically, I've noticed that the LanMan redirector returns failure for most of these requests. So, except for any user-defined FSCTL requests we want to define, I'm going to fail all of these until it turns out we need to do otherwise.

File System Set Info

Ditto for this IRP type too.

Volume Query Info

We at least need to minimally implement these requests:

- FileFsAttributeInformation
- FileFsVolumeInformation
- FileFsDeviceInformation

This will be handled solely by the EFSD; the request will not go out to the ECM. File system size requests will not be handled.

Volume Set Info

We will fail all requests of this type.

Flush Buffers

A buffer flush request for a file stream will mean the following:

- If the file stream isn't buffered, return immediately
- The FCB main Resource is acquired exclusive
- The Cache Manager is told to flush the buffer for the byte range of the file
- The resource is released

A buffer flush for a directory is a successful NOP.

The buffer flush request will insure that contents in the NT File Cache are written to the ECM (as a normal write request); the buffer flush request itself will not be propagated to the ECM.

Testing design

Unit testing plans

The EFSD will be tested apart from integrating with the ECM or the rest of the eStream client. Some points:

- There will be a (relatively) simple stand-in user process for the ECM, to get requests from the EFSD and handle them locally.
- As much EFSD functionality as possible will be done using user-mode test cases (e.g., open files, read/write files, delete files, etc.).
- Some functionality may need to be unit tested using another kernel-mode driver to send explicit IRPs to the EFSD.
- Filemon will be used to monitor the requests being handled by the EFSD.

Stress testing plans

I've heard of a file system filter driver test package available from Microsoft. This is probably the best way we have of stress testing the EFSD.

Coverage testing plans

We'll try to measure coverage on the EFSD. If there is a general kernel-mode method for measuring coverage that's used company-wide, we'll exploit it. Otherwise, there will be some primitive self-coverage instrumentation conditionally inserted in the driver code that we can use at least for major code paths.

Cross-component testing plans

There's not much to do here apart from normal interactions with the ECM.

Upgrading/Supportability/Deployment design

For supportability, there will be solid debugging aids—using printfs—built into the EFSD sources. Additionally, aside from good error codes returned from its interfaces, the EFSD will explore diagnostic traces optionally dumped for deployment.

Issues with stakes in the ground

- At present, I am assuming that there is a single drive letter associated with the EFSD, though there's no technical reason why this must be so. If indeed we organize the client system and the eStream file hierarchies to have multiple drive letters, either the EFSD or the ECM will need to parse the drive letter and do the right thing.
- This design assumes that 8.3 DOS-style filenames need not be supported. Adding such support will increase the complexity of the EFSD, as well as many other components in the eStream system: on the client, on various servers, and on the content builder.
- No support is provided in this design for:
    a. byte locks
    b. directory notification
    c. file open by file id
    d. file system control requests

Open Issues

1. I'm unclear on how to use the NT File Cache for metadata and directory contents. For now, I'm ignoring such matters, and we will only be caching file contents.
2. I do not know how to hook up the EFSD to UNC names. That is, I don't know how to set things up to have all file accesses like \\ASP1\Office\winword.exe be directed to the EFSD by the NT I/O Manager.
3. This design doesn't cover exactly how IRPs are posted for asynchronous processing. The SFSD example in Rajeev Nagar's book really isn't sufficient for some of what we need to do. Also, it's unclear to me what value there is to returning STATUS_PENDING and handling a request asynchronously if the caller is blocking anyway.


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back
Search
Folders
Address
develop/eng/estream/docs/client
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
2 Microsoft Ou...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
6:36 PM

[/STANDARD.01]

eStream 1.0 Prefetching & Fetching Low Level Design

Anne Holler *[redacted]* *Version 2.3*

Functionality

The EStream Prefetching and Fetching [EPF] module requests eStream application pages. The module handles demand fetching & prefetching. Prefetching is based on information in the prefetch section of the AppInstallBlock, on spatial locality, and potentially on profiling data gathered during previous runs of the application on this client.

Please see "eStream 1.0 Client Profile/Prefetch Straw Man" for background information related to prefetching & profiling. In particular, that document discusses the redundancy of gathering profiling information for prefetching on a client w/generous cache resources & the limitations of path-based prefetching when bandwidth is insufficient to hide latency without adequate execution-time lookahead. The collection & utilization of client profile information will be developed under an option (as it was for the prototype), and will be deployed in eStream 1.0 only if appropriate benefit is demonstrated.

Data type definitions

For core prefetching & fetching functionality, EPF uses the ECMRequest data structure.

For initial prefetch data, EPF reads FileNum,PageNum pairs from a file given by AIM.

For profiling data, EPF reads/writes FileNum,PageNum predecessor/successor per-AppID information from/to nonvolatile storage. This information is of a similar nature to the eStream cache and should be stored similarly on a per-client basis. For eStream 1.0, profile data is only stored if user has indicated that eStream cache can grow w/o a bound.

Interface definitions

From AIM to EPF:

- EPFPrefetchInitialAppBlocks(In AppID, In AppCorePrefetchSectionFileName, In AppOptionalPrefetchSectionFileName)

From ECM to EPF:

- EPFReportPage(In AppID, In FileNum, IN PageNum, In *Request)
- EPFReadPage(In AppID, In FileNum, In PageNum, In *Request)

From CUI to EPF:

- EPFPrefetchSet(In OnOff)

From EPF:

- LSMGetAccessToken(IN AppID, IN RequestHandle, IN ActivePrefetchOnly, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)
- ECMIsPageInCache(In AppID, In FileNum, In PageNum, In Priority, Out PageState)
- ECMReservePage(In AppID, In FileNum, In PageNum, In Priority)

Component design

Fetching and prefetching pages during program execution:

Interface with ECM [run in an ECM thread]:

```
ECM repeatedly calls EFSDGetRequest to pick up next eFSD request

EFSDGetRequest allocates ECMRequest data struct X & initializes

If the request is ERT_READ then

  ECM calls ECMReadPage w/ ECMRequest data struct X

  ECMReadPage acquires the work list/index lock

  If the READ hits in the cache then

    ECMReadPage calls EPFReportPage w/ AppId, FileNum, PageNum

    EPFReportPage allocates ECMRequest data structure Y,
       Puts the appropriate info including ERT_READ_HIT in it,
       Enqueues it in EPF's input queue & immediately returns

    ECMReadPage copies the data to the kernel and returns

  Else /* READ misses in the cache then */

    ECMReadPage calls EPFReadPage with AppID, FileNum, PageNum

    EPFReadPage allocates ECMRequest data structure Y,
       Puts the appropriate info including ERT_READ in it,
       Enqueues it in EPF's input queue & immediately returns

    /* EFSDCompleteRequest is called when data comes back from NW */

  End If /* READ hit or miss in cache */

  ECMReadPage releases the work list/index lock

Else if request is ERT_WRITE then

  /* For prototype, only prefetched & profiled based on READs */

End if /* Request is ERT_READ or ERT_WRITE */
```

EPF processing of its input queue [run in EPF thread]:

```
EPF repeatedly gets next ECMRequest in its input queue [using lock]:

Switch (RequestType)
  Case ERT_READ:
    Calls ECMReservePage with ECM Request data struct Y
    /* Fall through */
  Case ERT_READ_HIT:
    Call EPFRecordReference(In AppID, In FileNum, In PageNum)
    Call EPFPrefetchPredictedPath(In AppID, In FileNum, In PageNum)
    Break;
End switch
```

EPFPrefetchPredictedPath:

```
EPF predicts path based on current reference, on look-ahead distance,
  & potentially on previous references including stored profile data

For each page reference in the path

  EPF invokes ECMIsPageInCache

  If the page is not either in the cache or in flight then

    EPF calls ECMReservePage w/null ECMRequest (designates prefetch)

  End if

End for
```

EPFRecordReference:

```
EPF records pred/succ pairs of FileNum,PageNum on a per-AppID basis
```

Prefetching pages specified in AppInstallBlock:

```
At application install time, LSM calls ECMMount & then AIMInstall

AIMInstall calls EPFPrefetchInitialAppBlocks which returns immedi-
ately

EPF uses separate prefetching thread to obtain this app's core pages:

  For each (FileNum,PageNum) pair from AppCorePrefetchSectionFileName

    EPF invokes ECMReservePage w/priority parameter marked as criti-
cal
      and w/null ECMRequest (designates prefetch)

While cache utilization & client networking usage sufficiently low

  For each (FileNum,PageNum) from AppOptionalPrefetchSectionFileName
```

```
EPF invokes ECMReservePage w/null ECMRequest (designates pre-
fetch)
```

Prefetching pages independent of execution:

For client eStream installations with adequate caching resources, it may make sense to prefetch pages of files that are heavily-used & widely-covered. The winstone data indicated that such files include the main exe associated w/each application & certain core dlls. Hence, the idea of prefetching pages independent of eStream execution. This idea was not explored during the prototype phase & will be deployed in the eStream 1.0 product if it proves itself.

If the eStream client software is active for a particular user & it detects that the client system has been idle for an extended period of time, the independent prefetching system is engaged. The independent prefetching system decides which AppID would benefit from prefetching (somehow), which FileNum should have some of its pages prefetched (somehow), & invokes **LSMGetAccessToken** indicating ActivePrefetchOnly & specifying a callback routine to which LSM should report whether the AccessToken is obtained. ActivePrefetchOnly implies that the AccessToken should not be renewed automatically (to facilitate pre-emption of the AccessToken by a client who wants to use the app) and it implies that any problems getting the AccessToken should result in giving up rather than launching some dialog with the user. If the AccessToken is obtained, then as long as there are adequate cache resources & as long as the client network usage is low (aside from the bandwidth being used for prefetching!), EPF fetches pages in FileNum that are missing from the cache. The independent prefetching system disengages whenever client load detected, whenever an AccessToken is denied, whenever the cache is enhanced or polluted enough, etc.

Testing design

Unit testing plans/Coverage testing plans

A simple driver program, which will consist of a sequence of EPFReportPage & EPFReadPage calls and stubs for ECMIsPageInCache & ECMReservePage, will be used for unit testing of the runtime prefetching/fetching functionality. Expected profiling information will be compared with that collected and expected fetching/prefetching sequences will be compared with those produced. A dummy AppInstallBlock prefetch section file will be created and will drive unit testing of that portion of the prefetcher.

Stress testing plans

High fetching demand will be most stressful for this component. This may be tested by running many applications simultaneously on a client, forcing ECM to "miss" every reference, but having CNI return the data fairly quickly to keep the pressure on EPF.

Cross-component testing plans

This component is tightly coupled with the ECM. It will be integrated with the ECM as quickly as possible & will leverage off of ECM's testing infrastructure.

Upgrading/Supportability/Deployment design

A tracing feature will be introduced to have EPF report each fetch & prefetch it is making and why. An option to disable prefetching (and potentially profiling) will be provided.

eStream 1.0 Client Profile/Prefetch Straw Man

*Anne Holler *[redacted]* Version 1.0*

Introduction

This document presents background issues related to the collection of eStream client profile data & its use in client prefetching and client cache replacement policy and lists proposed design decisions wrt client profiling and prefetching for eStream 1.0. The document is intended to provide a baseline for discussions prior to proceeding with a detailed low level design.

Client Profile Data Background

eStream profile data may be used in client prefetching & client cache replacement policy. Ideally, eStream profile data consists of a sequence of (FileNum,PageNum) entries which comprise all references made on behalf of some run(s) of a particular eStream app, with the idea that any of these references would potentially result in an eFS request. Dynamic page tracing based on page faults gives a subset of the ideal trace, which includes the page references for compulsory misses, some subset [possibly empty] of the page references for the non-compulsory misses, and no entries for the remaining page references. A recently proposed augmentation of this data with information from static analysis of executables misses indirect links between pages and is in general imperfect; subprogram entry points will not always be visible and code & data may be interleaved on the same page in x86 code. Gathering ideal eStream profile data is a heavy-weight process, involving instruction level tracing.

What kinds of issues could arise with respect to incomplete eStream profile data? To illustrate, let us assume that the sequence {b,c,d,x,y,z} appears in an incomplete eStream profile as {b,z}. One issue is that we do not know how far ahead a page in an incomplete profile will actually be used; prefetching z at the time b is being accessed is a problem if z displaces any of {c,d,x,y}. Another issue is that we may miss prefetch opportunities; the incomplete profile does not indicate that we need to prefetch {c,d,x,y} if accesses to them page fault and the pages are not in our cache. Another issue is that incomplete profile data does not allow us to know which pages are used frequently; if b is used frequently, we might see only one reference to it per run in the ordered profile data (because its frame tends to stay in main memory), whereas if z is used infrequently, we might see no references to it on some runs and possibly multiple references to it on other runs (assuming its infrequent use means its frame tends to get displaced from physical memory). Hence, straightforward use of either reference counts or LRU based on incomplete eStream profile data can be flawed wrt client cache replacement.

If incomplete profile data based on page faults is both collected and used on the same client, the issues cited above are less likely to occur (assuming the execution load on the client is roughly the same each time the application is invoked); profile data collected on an eStream builder machine and downloaded to a customer client machine at install time would be expected to be more likely to encounter these issues. While various approaches have been discussed to mitigate the potential impact of these problems, the decision has been made to forgo the profile section of the AppInstallBlock for eStream 1.0 (largely because the planned lightweight builder training strategy means that the profile section contents would be redundant with the prefetch section contents). Uploading client profile data, which has previously been discussed, is also outplan for eStream 1.0; since there is no initial downloaded profile data, there is no need to use uploaded client profile data to refine it. Application usage data, which may be of interest to ISVs and others, will be gathered at a less granular level from the SLM server.

What good is page-fault-based client profile data collected & used on the same client? If the client has a cache of unlimited size, in which no block is ever replaced, this profile data is redundant in the sense that it brings no obvious value to eStream 1.0 in addition to what is already provided by the contents of the client cache. If the client cache does need to do replacement, this profile data can be used to guide the replacement strategy, but only if the data is used in a clever way to avoid the oddities that result from its incompleteness (e.g., use per run); if it is not used cleverly, one may as well use random replacement. Client cache replacement implies that refetching of displaced pages may be needed. The profile data can potentially do a good job of supporting prefetching of this refetching; in experiments with the prototype, one can delete a warmed client eStream cache, just use the profile data, & hide virtually all latency associated with the refetches, at least in the scenario in which there is adequate bandwidth. For a 100Mbps link, executing start/exit of word, cache hit rate with profile-guided prefetching is 98%, with 6% additional overhead for redundant prefetch.

How does profile-based prefetching compare with simple locality-based prefetching, for which no special technology is needed to collect and use the profiles is needed? In experiments with the prototype, locality-based prefetching yielded not-too-terrible results as compared with profile-based prefetching wrt client cache hit rate, but at the cost of a much higher wasted prefetch rate, i.e., at the cost of much more network bandwidth and server traffic. For 100Mbps link, executing start/exit of word, cache hit rate with locality-guided prefetching is 75%, with 89% additional overhead from redundant prefetch. We need to decide if client profiling is worthwhile for eStream 1.0, based on whether we believe there will be sufficient cache replacement and if so, whether we want to avoid the extra overhead of locality prefetching to restore displaced pages and if there will be sufficient bandwidth to exploit the extra knowledge that profile data gives us [perhaps via idle prefetch]. Putting a stake in the ground, I advocate designing and building client profile gathering and associated prefetching, because I believe it may be useful and because I believe we will be able to learn things from it which may profit us in the future. Please note that for the prototype, client profile gathering was under an #ifdef, so it could be removed as desired, since it adds memory consumption and overhead to the eStream client. We could develop this way for the real product as well, in case we decide

that it is not worthwhile for the eStream 1.0 product at all or at least for some particular variant of the product which is geared to very fat cache clients.

Please note that, in general, client profiling is done on a per application basis; if the user is running several eStream applications on his client, it is expected that we would want to learn page sequences within each application, not sequence relationships across apps. Similarly, client profiling is done on a per invocation basis; if the user is running several instances of a particular eStream application, it is expected that we would want to learn about the sequences belonging to each instance separately. For the prototype, we turned off learning profile sequences if more than one instance of an application was running; we may want to consider doing this for the real product as well.

Client Prefetching Background

When does a client do eStream prefetching? The most intuitive time to do eStream client prefetching is at eStream app runtime, in association with runtime behavior; this is the only time at which prefetching is done in the prototype. Two other times to consider doing eStream prefetching are at app installation time and at network idle time. At app installation time, the AppInstallBlock contains a prefetch section, which specifies {FileNum,PageNum} pairs for pages which are recommended to prewarm the client cache for this app; the client prefetcher will ask the client networking code to request these pages from the appropriate app server. At network idle time (i.e., when there is little client networking traffic), app pages may be downloaded without directly impacting client eStream performance; clearly such prefetching would need to be done judiciously to avoid polluting the client's cache, impacting server scaling, wasting network bandwidth, or hogging the app's license, but this could be a interesting strategy for eStream technology if it were done effectively.

What does the eStream client prefetch? As discussed in the previous section on client profiling, profiling info can be used to drive which pages to prefetch, either by prefetching along an observed path or by prefetching according to a projected pattern, possibly based on observed references. Spatial locality is a frequently observed pattern in general, and may be used as the default projected pattern in the absence of relevant profiling information. In eStream 1.0, we may also consider using semantic information to drive prefetching. One example might be prefetching the contents of the directory files for an installed app, since directory files may contain useful FileIDs and metadata. Another example might be to prefetch any missing pages of heavily used files such as the main executable for each application, with the idea that the pages might eventually be used for some app feature not yet exercised. Several folks have suggested that we gradually download the entire app, if cache space allows, with the vision that this provides full native performance; if this can be done without impacting any of our core selling points, this might be worth considering.

Client Profile Data Collection Design Proposal

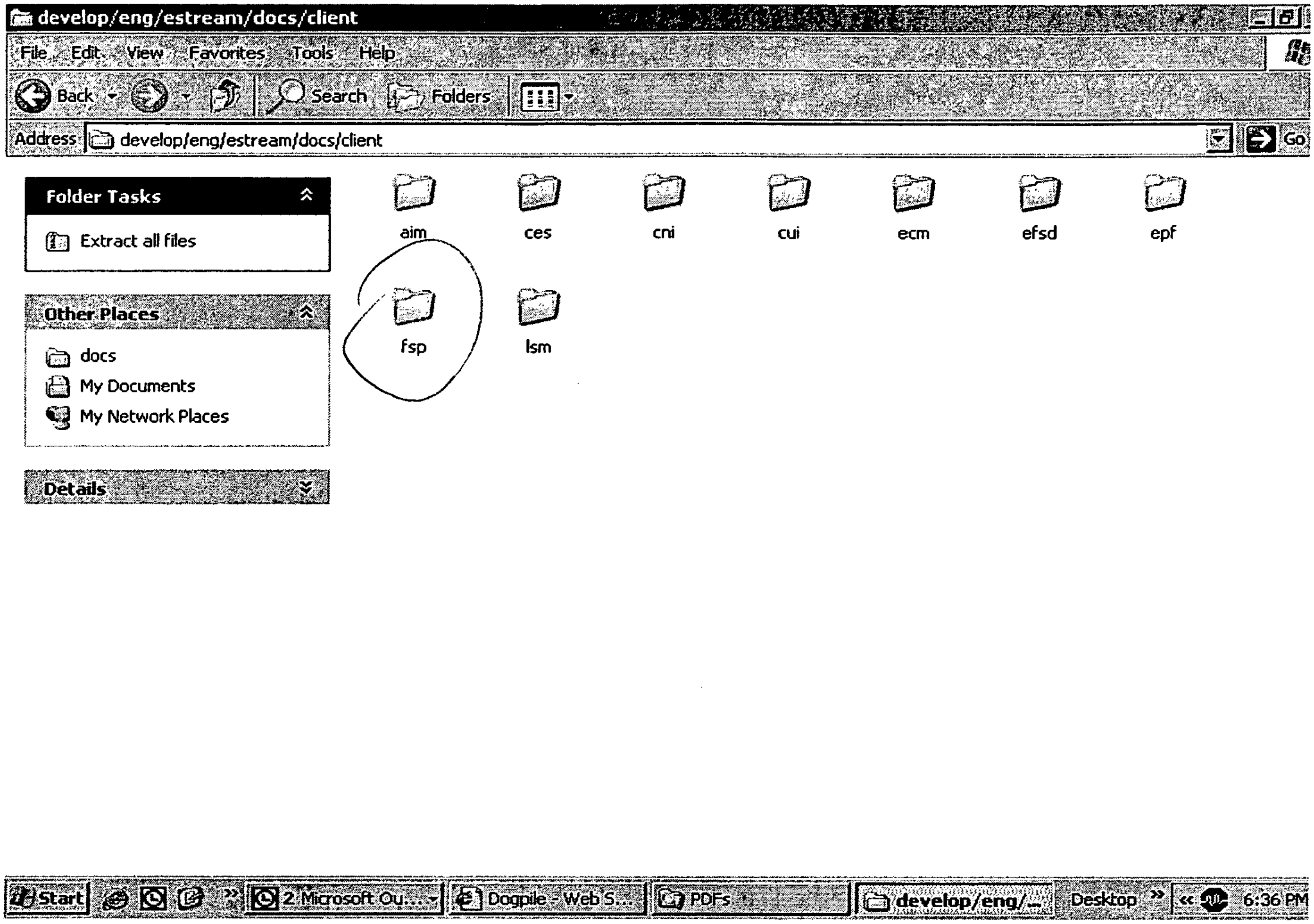
Here are key points on the design of eStream 1.0 client profile data collection:

*) Have profiler see all requests coming from eFSD
*) Record profile data on per-application basis
*) Record profile sequence for app only if single instance of app running (how can tell?)
*) Represent (potentially lengthy) profile data list (with some loss) as (pred,succ) tuples
*) Make client profiler optional
*) Mark use-per-run pages high priority to aid cache replacement policy

Client Prefetching Design Proposal

Here are key points on the design of eStream 1.0 client prefetching:

*) Prefetch at app install, mark specified pages as high priority to avoid their replacement
*) Also prefetch at app runtime, have prefetcher see all requests coming from eFSD
*) If eFSD page request on known path, project next refs on path; else, use spatial locality
*) Prefetch only projected refs that are not already in client cache
*) Consider network speed, path likelihood, cache issues in determining prefetch distance
*) Also prefetch at idle, if unlimited cache; get missing pages from app's main executable


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back
Search
Folders
Address
develop/eng/estream/docs/client
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
2 Microsoft Ou...
Dogpile - Web S...
PDFs
develop/eng/...
Desktop
6:36 PM

eStream File Spoofer Low Level Design

Curt Wohlgemuth
Version 2.0
[redacted]

Functionality

The eStream file spoofer is a kernel-mode driver responsible for redirecting file accesses **from** local file systems **to** the eStream file system driver. It is implemented as a file system filter driver that traps all IRP requests to the file system device handling drives that must be spoofed, and redirects these requests to the EFSD.

Data type definitions

The file spoofer will understand entries in the "file spoof database" as they have been identified by the eStream builder and installed by the app install manager, but these are not defined by this component.

Entries in this spoof database will have the following entries:

- original (fully qualified) path name of file: this resides somewhere on a local disk of the client machine
- new (fully qualified) path name of the file to spoof to: this resides on the eStream file system drive

The spoof database will reside in the registry, so it can be persistent across reboots, and so the file spoofer need not open a file to load it. As applications are installed on a client machine, the Application Install Manager will load new spoof entries into the registry, and inform the file spoofer that it must reload this database. Similarly, when an app is uninstalled, the AIM will remove spoof entries from the registry, and inform the file spoofer.

This proposes that the each spoof entry is a separate name/value entry under a single key in the registry:

- Name: the original filename
- Value: the new filename

Interface definitions

The eStream file spoofer is called by two components:

1. The eStream client start service, which will start and stop the file spoofer

2. The AIM, which will inform the file spoofer to reload the spoof database from the registry

The interfaces called by both of these user-mode components will be in the form of DeviceIoControl() calls. The following IOCTL codes will be defined for use by callers of the file spoofer:

```
IOCTL_FS_START_SPOOFING
IOCTL_FS_STOP_SPOOFING
IOCTL_FS_RELOAD_SPOOF_DB
```

Starting spoofing

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that starting spoofing is currently identical to reloading the spoof database.

This is called by the eStream client start service on startup.

Stopping spoofing

The input and output buffers for this IOCTL are ignored and should be NULL. This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to clear its memory image of the spoof database.

This is called by the eStream client start service on shutdown.

Reload spoof database

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that this is currently identical to starting the spoof database.

This is called by the AIM when a new eStream app is installed.

Component design

The file spoofer will have these major tasks:

- Track the following data:
  - all current valid file spoof entries
  - spoof entries by filtered file system
- Filter native file system drivers for local drives, intercept all IRP_MJ_CREATE and FAST_IO_QUERY_OPEN requests, and for spoofed files, change the filename of the FileObject associated with these requests.

Data structures

The file spoofer needs to be able to quickly look up a filename in the in-memory spoof database. The current design will use a hash table, whose size and hash function will be tuned as we get experience with real applications.

Adding or deleting entries from the hash table will by synchronized using a global resource.

Algorithms

Here are basic algorithms for these steps.

Load spoof database

This reads all the name/value pairs under the registry key which holds the spoof entries, loads them into a temporary hash table, then points the real hash table to this one.

```
traverse the registry until the input registry key is opened, using ZwOpenKey()
if not found
    return error
if no name/value pairs exist in this key
    return "no data"

for each name/value pair found with ZwEnumerateValueKey()
    build a hash node for this and insert it into temp hash table
    if the drive letter for the old filename entry is one we're not currently filtering
        put this drive letter on new drive list

acquire the hash table resource exclusively
point the global hash table head pointer to the temp hash table
release the resource

for each drive letter on the new drive list
```

```
        look up FS device for this drive
        if we really aren't attached to it
            attach self to this FS device as filter driver

    free the old hash table
    free the drive list
    return success
```

Stop spoofing

```
    acquire hash table resource exclusively
    free the global hash table
    detach self from all filtered FS devices
    release hash table resource
```

Trap Create and QueryOpen requests

```
    acquire the hash table resource shared
    if hash table head pointer is non-NULL
        look up input filename in hash table
    release hash table resource
    if filename not found in hash table
        send I/O request to original target FS driver
    else
        free memory of existing file name in input FileObject
        allocate memory for new, spoofed filename, copy into this memory
        send I/O request to eStream file system driver
```

Testing design

Unit testing plans

The file spoofer will be tested as a standalone component, apart from the rest of the eStream client. A driver test program will be written to test all functionality and corner cases. This includes filtering all FSDs active for a client system, and multiple drives handled by a single FSD.

Stress testing plans

The file spoofer should be able to work, with little or no performance cost to the system as a whole, even when the attached FSDs are under heavy load. Some stress testing will be done in this fashion.

Coverage testing plans

If we come up with a method for measuring coverage for kernel-mode components, we'll do so for the file spoofer as well.

Cross-component testing plans

Not clear if anything need be done here outside of the standard execution of the eStream client.

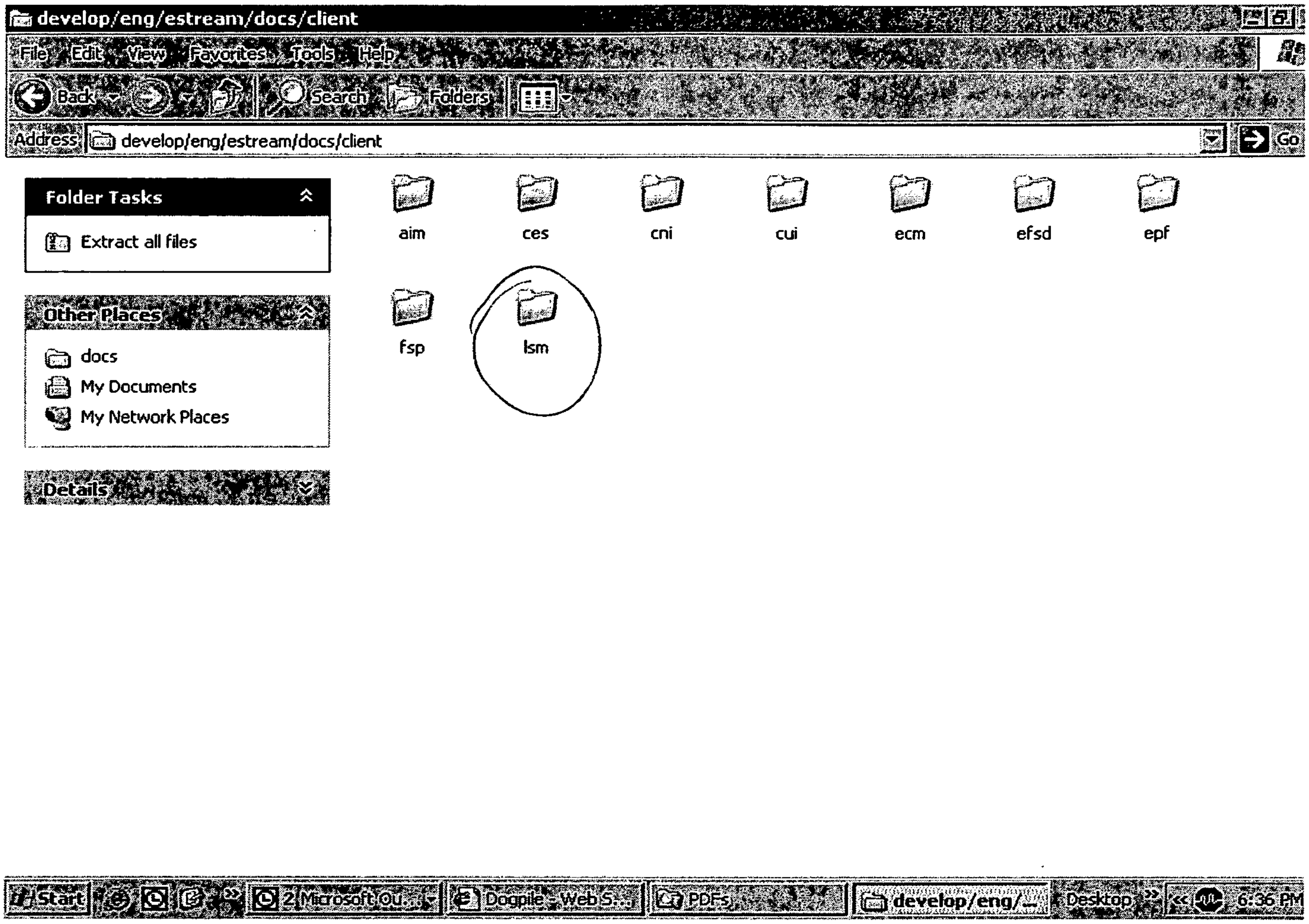
Upgrading/Supportability/Deployment design

I don't see any upgrade/compatibility issues for the file spoofer. For supportability, there will be a good debugging strategy and sufficient error message return codes for the caller.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. We will need to experiment with the hash table to tune it for fast lookup. It's possible that we may need to replace the hash table with a faster lookup algorithm.



[/STANDARD.01] [redacted]

eStream 1.0 Client License Subscription Mngr Low Level Design

Anne Holler * [redacted] * *Version 2.9*

Functionality

This document presents the low level design of the License Subscription Manager [LSM], an eStream client module that tracks information about ASP accounts, application subscriptions, and installed applications. The document also describes the interface to the eStream Client Browser Module [CBM], a client component that allows an ASP web server to notify LSM of client-relevant changes in an ASP account. (The CBM interface is used in specific situations, as described below; ASP account changes are detected automatically in steady-state operation.)

Data definitions

There are two data sets that LSM maintains on a per-user basis on the client [*italics* => only volatile storage]. One set is associated with the ASPs from which the client eStreams applications & the other is associated with the eStream applications to which the client is subscribed. This data is kept in the registry.

ASP Data Set

- uInt128 AspID -- Local handle for client convenience
- string UserName
- *string Password* – Only volatile storage unless user approves nonvolatile
- string ASPWebServerName -- May simply be an ip address
- eStreamServerSet SLiMServerSet

Subscription Data Set

- uInt128 SubscriptionID
- uInt128 AppID
- Int32 RootFileNumber
- *eStreamServerSet AppServerSet*
- *eStreamAccessToken AccessToken*
- *uInt64 ExpectedExpirationTime*
- enum {Installed, NotInstalledPrompt, NotInstalledDontPrompt} AppInstallStatus
- enum {Updated,NotUpdatedPrompt,NotUpdatedDontPrompt} AppUpgradeStatus
- enum {ActiveInUse, ActivePrefetchOnly, Inactive} AppActiveStatus
- string AppName
- string VersionName
- string Message
- uInt128 AspID -- Local handle

Interface definitions

Client Component Acronyms

- AIM: Application Installation Manager
- CBM: Client Browser Module
- CES: Client EStream Startup
- CNI: Client Network Interface
- CUI: Client User Interface
- ECM: EStream Cache Manager
- LSM: License Subscription Manager

LSM InterfacesTo/From Server Components

From LSM to SLiMServer via CNI:

- CNIGetSubscriptionList(IN AspID, IN UserName, IN Password)
- CNIGetLatestApplicationInfo(IN AspID, IN SubscriptionID)
- CNIAcquireAccessToken(IN AspID, IN SubscriptionID, IN UserName, IN Password)
- CNIRenewAccessToken(IN AspID, IN SubscriptionID, IN UserName, IN Password, IN AccessToken)
- CNIReleaseAccessToken(IN AspID, IN SubscriptionID, IN UserName, IN Password, IN AccessToken)
- CNIRefreshAppServerSet(IN AspID, IN SubscriptionID IN AccessToken, IN BadQos, IN NoService)

From ASPWebServer to CBM:

- ASPNotify(IN UserName, IN SLiMServerSet)

From LSM to ASPWebServer:

- PromptASPNotify: HTTP POST to server cgi script to engender an ASPNotify

LSM Interfaces To/From Client Components

From ECM to LSM:

- **LSMGetAccessToken**(IN AppID, IN RequestHandle, IN ActiveInUse, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)
- **LSMReleaseToken**(IN AppID)

From LSM to ECM:

- ECMMount(IN AppID, IN RootFileNumber)
- ECMRemount(IN AppID, IN OldRootFileNumber, IN NewRootFileNumber)
- ECMSetTokenState(IN AppID, IN State) – *NotHolding, Holding, Denied*

From CBM to LSM:

- **LSMAspAccntPing**(IN AspWebServerName, IN UserName, IN SLiMServerSet)

From CES to LSM:

- **LSMInitialize**(IN WindowsUserName) -- Read ASP & Subscription Data Sets from registry
- **LSMUpdateAllSubscriptionStatus**(void)

From LSM to AIM:

- AIMInstall(IN AppID, IN AppInstallBlockFileName, OUT InstallStatus)
- AIMUninstall(IN AppID, OUT UninstallStatus)

From LSM to CUI:

- CUIInformUser(IN Message)
- CUIAskUserYesNo(IN Message, IN CheckAskAgain, OUT Response, OUT DontAskAgain)
- CUIAskUserPassword(IN Message, OUT Response, OUT Retain)

From CUI to LSM:

- **LSMUpdateAllSubscriptionStatus**(void)
- **LSMGetAspList**(OUT NumAsps, OUT AspID[])
- **LSMGetAspInfo**(IN AspID, OUT ASPWebServerName, OUT UserName)
- **LSMGetAppList**(IN AspID, OUT NumApps, OUT AppID[])
- **LSMGetAppInfo**(IN AppID, OUT AppName, OUT VersionName, OUT Message, OUT AppInstallStatus, OUT AppUpdateStatus)
- **LSMInstall**(IN AppID, OUT InstallStatus)
- **LSMUninstall**(IN AppID, OUT UninstallStatus)
- **LSMUpgrade**(IN AppID, OUT UpgradeStatus)

From EPF to LSM:

- **LSMGetAccessToken**(IN AppID, IN RequestHandle, IN ActivePrefetchOnly, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)

From CNI to LSM:

- **LSMGetAppServerSet**(IN AppID, OUT AppServerSet)
- **LSMGetNewAppServerSet**(IN AppID, OUT AppServerSet)
- **LSMGetSLiMServerSet**(IN AspID, OUT SLiMServerSet)
- **LSMGetNewSLiMServerSet**(IN AspID, OUT SLiMServerSet)
- **LSMGetAccessToken**(IN AppID, IN RequestHandle, IN Unknown, IN RegisterCallback, OUT AccessTokenStatus, OUT AccessToken)
- **LSMReturnSubscriptionList**(IN AspID, IN ReturnCodes, IN NumberOfSubscriptions, IN SubscriptionID[])
- **LSMReturnLatestApplicationInfo**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN UpgradeInfo)
- **LSMReturnAcquireAccessToken**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN AccessToken, IN RenewalFreq, IN AppServerSet, IN UpgradeInfo)
- **LSMReturnRenewAccessToken**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN AccessToken, IN RenewalFreq, IN AppServerSet)
- **LSMReturnReleaseAccessToken**(IN AspID, IN SubscriptionID, IN ReturnCodes)
- **LSMReturnRefreshAppServerSet**(IN AspID, IN SubscriptionID, IN ReturnCodes, IN ServerSet)

Component design

The LSM is a service module that handles requests from eStream client software and from the ASPWebServer via the CBM [discussed under its own heading at the end of this section]. Hence, LSM's component design is presented here in terms of how it responds to the requests that it receives in the three main scenarios in which it participates: running an eStream application, getting/refreshing application subscription information, and displaying subscription information.

The web pages at \\fserv2\home\webpages\pmain.html are intended to provide a framework for discussing overall issues related to subscribe/install/uninstall scenarios. The discussion of scenarios in this section is at a much lower level than that material.

Run an Application

- ECM invokes **LSMGetAccessToken** whenever there is a reference to a file belonging to a particular AppID for which ECM's internal data structure indicates that the AccessToken is in the *NotHolding* state. [ECM can determine the AppID of an eStream app by inspecting the fully-specified eFSD filename; the first level of the directory path is a string version of the AppID. If LSM does not recognize this AppID for some reason, an appropriate status is returned to ECM.]
- LSM marks the AppID as ActiveInUse in the Subscription Data Set & returns the appropriate status to ECM. If the status indicates that LSM has an AccessToken for the application, ECM updates its internal data structure to indicate the AccessToken is in the *Holding* state & continues. If the status indicates that LSM

does not have an AccessToken for the application, LSM takes ownership of the specified request, holding the RequestHandle in an internal queue for an eventual reissue request to eFSD.

- LSM issues a **CNIAcquireAccessToken** call via CNI to a SLiMServer & when LSM receives the AccessToken via the **LSMReturnAcquireAccessToken** call-back, it places a copy of it in the volatile Subscription Data Set, assigning ExpectedExpirationTime according to the RenewalFrequency indicated by the SLiMServer and setting up a timer alarm to go off shortly before the AccessToken is expected to expire. It calls **ECMSetTokenState** to notify ECM that the AccessToken is in the *Holding* state for the AppID.
- LSM compares the application's current RootFileNumber on the client with UpgradeInfo.RootFileNumber returned wrt the **CNIAcquireAccessToken** call; if the latter is higher & UpgradeInfo.ForcedUpgrade is off, it calls **CUIAskUserYesNo** about upgrading. LSM sets AppUpgradeStatus and RootFileNumber fields in the Subscription Data Set appropriately & calls **ECMRemount** if there is a minor upgrade.
- Should an AccessToken be denied due to a mandatory major upgrade that costs money, LSM calls **CUIInformUser** to let him know that he needs to interact with his ASP to revise his account.
- Should an AccessToken be denied due to a mandatory major upgrade that does not cost money, LSM calls **CUIAskUserYesNo** about upgrading. If he responds affirmatively, the Subscription Data Set is updated, **AIMUninstall** is called to nuke the current version, & **AIMInstall** is called to install the upgraded version.
- Should an AccessToken be denied because it is currently held by another of the user's clients, a copy of the other client's AccessToken is returned. If the user responds affirmatively to **CUIAskUserYesNo("Yank?")**, then LSM will call **CUIInformUser("Please wait")**, send the token to **CNIReleaseAccessToken,** & then retry **CNIAcquireAccessToken** at the appropriate time.
- For AccessToken denials that stick [either because the user has decided not to take a mandatory upgrade at this time, or because the user has decided not to yank another of his clients' AccessToken, or because the user has some problem with his ASP account (didn't pay, account suspended/terminated, got evicted, authorization failed)], the user is informed of the problem and told to save/exit the application ASAP. **ECMSetTokenState** is called to mark the token state *Denied* and EFSD is told to retry the request. When ECM's internal data structure has the AccessToken status for an application marked *Denied*, it allows the user to continue running out of the cache, but will report failure to EFSD immediately on any access that misses in the cache.
- CNI frequently invokes **LSMGetSLiMServerSet** to obtain SLiMServers for AccessToken requests. Should CNI have quality of service issues with those servers, it calls **LSMGetNewSLiMServerSet**; LSM uses the **PromptASPNotify** interface to engender the ASP Web Server to send a message containing a SLiMServerSet.
- Whenever an AccessToken timer alarm goes off, LSM checks that the associated app is marked ActiveInUse & if so, it issues a **CNIRenewAccessToken** & puts token returned by **LSMReturnRenewAccessToken** in the Subscription Data Set, resetting ExpectedExpirationTime and scheduling another timer alarm.

- When CNI needs to fetch requested data from an AppServer, it invokes **LSMGetAccessToken**.
- CNI frequently calls **LSMGetAppServerSet** to obtain a set of AppServers through which to cycle; LSM returns the servers associated with the most recently acquired AccessToken. Should CNI be dissatisfied with the quality of service associated with the AppServers it is given, it calls **LSMGetNewAppServerSet** which sends **CNIRefreshAppServerSet** to a SliMServer and receives a **LSMReturnRefreshAppServerSet** callback.
- Should a CNI request be denied by an AppServer because the submitted AccessToken was expired, CNI calls **LSMGetAccessToken**, which operates as described above with respect to acquiring the AccessToken and with respect to sending a reissue request to eFSD.
- ECM invokes **LSMReleaseToken** whenever there is a transition from 1 to 0 open files wrt a particular eStream app, setting the AccessToken status in its internal data structure to *NotHolding* (from either *Holding* or *Denied*). LSM sends a **CNIReleaseAccessToken** call to a SLiMServer [possibly only if the ExpectedExpirationTime for the AppID in question is not in the past] & receives an **LSMReturnReleaseAccessToken** callback. LSM marks the application as Inactive in the Subscription Data Set, indicating that automatic AccessToken renewal should not be continued for this application [LSM may also cancel any outstanding alarm].

Get/Refresh Subscription Information

- Whenever the CBM receives an **ASPNotify** from an ASPWebServer, it sends an **LSMAspAccntPing** message to the LSM. The LSM checks its ASP Data Set to see if the specified ASP is one that it recognizes. If LSM does not recognize the ASP, it allocates a new ASP Data Set entry and calls **CUIAskUserPassword** to prompt the user for the password associated with this ASP+UserName (asking the user if the password can be saved on the client for future use). If LSM does recognize the ASP, it only calls **CUIAskUserPassword** if the user did not allow the password to be previously recorded.
- If the user indicates that his password can be saved on the client for future use, we encrypt the password using a reasonably secure symmetric algorithm, such as DES, the key being a hash of the AspID and the username. If we implement the eStreamClientExecutable as a service (outstanding implementation issue in CUI LLD), we store the password in a registry key that cannot be read by ordinary user accounts (services can run in the context of a separate account), else we store the password in a registry key belonging to the user.
- Once LSM has the UserName & Password, it issues a **CNIGetSubscriptionList** request to one of the ASP's SLiMServers. For any SubscriptionIDs associated with that ASP that are installed on the client but do not appear in the list returned by the **LSMReturnSubscriptionList** callback, the user is informed that he cannot run the app at the current time & is asked if he would like it uninstalled from his client (note that we may be legally required to remove it from the cache, so this

may not be a question but rather just a message); **AIMUninstall** is invoked to achieve this.

- For all other SubscriptionIDs, a **CNIGetLatestApplicationInfo** request is sent to a SliMServer & a **LSMReturnLatestApplicationInfo** callback provides the data.
- For SubscriptionIDs already in the Subscription Data Set, the RootFileNumber is compared to see if the application has been upgraded. If it has, the user is informed; the new RootFileNumber is recorded (assuming either that the upgrade is mandatory or that the user indicated he wanted it) and **ECMRemount** is called to notify ECM about the upgrade so that it can take appropriate action. [Please note that for eStream 1.0, we will not support minor upgrades that involve new AppInstallBlocks; if a new AppInstallBlock is needed, the upgrade is by definition a major upgrade & the app needs to have a new AppID assigned & uninstall/install actions taken.]
- For SubscriptionIDs that were not already in the Subscription Data Set, a new Subscription Data Set entry is created and **CUIAskUserYesNo**("Install App?") is invoked. If user responds in the affirmative, LSM requests an AccessToken, calls **ECMMount** to notify ECM to prepare to service requests on behalf of this new application & then invokes **AIMInstall**, which can use the eFSD/ECM path to obtain the AppInstallBlock. If user responds in the negative, he is able to indicate "don't ask again"/"ask again later", which is stored in the Subscription Data Set.
- Whenever the eStream client software starts up or at specific times requested by the user via the ClientUI, **LSMUpdateAllSubscriptionStatus** is called to refresh the eStream client's knowledge of the apps to which the user is subscribed.

Display Subscription Information

- The CUI provides a user interface to display ASP and application info, using **LSMGetAspList, LSMGetAspInfo, LSMGetAppList, LSMGetAppInfo.**
- Please see "eStream 1.0 Client User Interface Low Level Design" for information on **LSMInstall, LSMUninstall**, and **LSMUpgrade.**

Client Browser Module Overview

The CBM is an eStream client component that runs in association with the client browser. CBM communicates a small amount of data to the core eStream client software (as described above) and it is realizable on both Internet Explorer 4+ and Netscape Navigator 4+. It does not have to start the eStreamClientExecutable if it is not already running.

Testing design

Unit testing plans/Coverage testing plans

For unit testing of LSM, a simple driver program will be written which will simulate the three main scenarios highlighted in this design document. It will exercise all LSM external entry points (listed in bold in the Interface Definitions section of this document) and

will achieve ~100% PFA coverage of all non-assert paths as measured by Rational Pure-Cov. A process will be introduced to automate the execution of this testing.

Cross-component testing plans

As defined in the high-level design, LSM has many cross-component interfaces. In the steady-state operation of eStream, its most heavily-used interfaces will be those with ECM and CNI. It is anticipated that LSM will be integrated first with these two pieces and will be added to an automated nightly testing of running eStream applications.

At eStream client software activation time, the CES interface is exercised; integration with CES and automated testing of the pair should be deployed as soon as both are available. At application install and uninstall time, the AIM interface is utilized; presumably, automated testing of installation & uninstallation of apps will be key to the AIM testing strategy and can be combined with the testing of this module when the two are integrated.

The CUI has a number of interfaces to LSM. Automated testing of the pair will be based on a UI testing tool like Rational's Visual Test tool (which BTW drives Winstone).

Stress testing plans

The most stressful usage scenario for LSM is high client load, i.e., many eStream applications running simultaneously. Visual Test will be used to set up this kind of testing scenario.

Upgrading/Supportability/Deployment design

LSM is heavily involved in user scenarios involving upgrading, supporting, and deploying eStream applications. The normal operation of these scenarios is automatic and seamless, but all are intended to have CUI interfaces that will allow manual intervention and troubleshooting.

In addition, LSM will provide a debugging interface for dumping the in-memory images of the ASP Data Set and the Subscription Data Set.

Implementation Issues

- Develop mechanism for implementing CBM. The plan is to make CBM a browser helper application, i.e., a standard windows32 app that is registered to handle a particular MIME encoded filetype (*.estream) in the HTML stream. [BTW, this is done by SoftwareWow.] As an alternative proposal, Bhaven created JavaScript that starts Excel & communicates with it, though this solution seems to require that the browser be Internet Explorer, that the app be ActiveX, and that the security setting on the browser be at minimum setting.

- LSM may move to kernel space and its implementation will avoid constructs which might unduly complicate this potential migration.

eStream 1.0 Client License Subscription Manager Straw Man

*Anne Holler * [redacted] * Version 1.2*

Introduction

This document presents background information related to the License Subscription Manager [LSM], an eStream client module that tracks information about ASP accounts, application subscriptions, and installed applications. The document also describes the interface to the eStream Client Browser Module [CBM], a client component that allows an ASP web server to notify LSM of client-relevant changes in an ASP account. (The CBM interface is used in specific situations, as described below; ASP account changes are detected automatically in steady-state operation). The document is intended to provide a baseline for discussions prior to proceeding with a detailed low-level design.

LSM Data Sets

There are two data sets that LSM maintains on the client in nonvolatile memory. One set is associated with the ASPs from which the client eStreams applications and the other is associated with the eStream applications to which the client is subscribed.

ASP Data Set

uInt128 AspID -- Local handle for client convenience
string UserName
string Password
string ASPWebServerName
eStreamServerSet SLiMServerSet

Subscription Data Set

uInt128 SubscriptionID
uInt128 AppID
Int32 RootFileNumber
eStreamServerSet AppServerSet
eStreamAccessToken AccessToken
uInt64 ExpectedExpirationTime
enum {Installed, NotInstalledPrompt, NotInstalledDontPrompt} AppInstallStatus
enum {ActiveInUse, ActivePrefetchOnly, Inactive} AppActiveStatus
string AppName
string VersionName
string Message
uInt128 AspID -- Local handle

LSM Interfaces To/From Client Components

The use of the interfaces listed below is discussed in the **LSM Functionality Overview** section of this document; the interfaces are summarized here for easy access.

Client Component Acronyms
AIM: Application Installation Manager
CBM: Client Browser Module
CES: Client EStream Startup
CNI: Client Network Interface
CUI: Client User Interface
ECM: EStream Cache Manager
LSM: License Subscription Manager

From ECM to LSM:
LSMNotifyAppStart(IN AppID, OUT AppStatus, OUT RootFileNumber)
LSMNotifyAppStop(IN AppID)
LSMCheckAccessToken(IN AppID, OUT AccessTokenStatus)
LSMGetAccessToken(IN AppID, OUT AccessToken)

From LSM to ECM:
ECMMount(IN AppID, IN RootFileNumber)

From CBM to LSM:
LSMAspAccountPing(IN AspWebServerName, IN UserName, IN SLiMServerSet)

From CES to LSM:
LSMUpdateAllSubscriptionStatus(void)

From LSM to AIM:
AIMInstall(IN AppID, IN AppInstallBlockFileName, OUT InstallStatus)
AIMUninstall(IN AppID, OUT UninstallStatus)

From CUI to LSM:
LSMUpdateAllSubscriptionStatus(void)
LSMGetAspList(OUT NumAsps, OUT AspID[])
LSMGetAspInfo(In AspID, OUT …)
LSMGetAppList(In AspID, OUT NumApps, OUT AppID[])
LSMGetAppInfo(In AppID, OUT …)

From LSM to CUI:
AskUserYesNo(IN Message, OUT Response)
AskUserPassword(IN Message, OUT Response, OUT Retain)
InformUser(IN Message)

From CNI to LSM:

LSMGetAppServerSet(IN AppID, OUT AppServerSet)
LSMGetNewAppServerSet(IN AppID, OUT AppServerSet)
LSMGetSLiMServerSet(IN AppID, OUT SLiMServerSet)
LSMGetNewSLiMServerSet(IN AppID, OUT SLiMServerSet)

LSM InterfacesTo/From Server Components

From ASPWebServer to CBM:

ASPNotify(IN UserName, IN SLiMServerSet)

From LSM to ASPWebServer:

PromptASPNotify: HTTP POST to server cgi script to engender an ASPNotify

From LSM to SLiMServer via CNI:

```
GetSubscriptionList(IN UserName, IN Password,
                    OUT ReturnCodes, OUT NumberOfSubscriptions,
                    OUT SubscriptionID[])
GetLatestApplicationInfo(IN SubscriptionID,
                         OUT ReturnCodes, OUT UpgradeInfo)
AcquireAccessToken(IN SubscriptionID, IN RootFileNumber,
                   IN UserName, IN Password,
                   OUT ReturnCodes, OUT AccessToken, OUT RenewalFreq,
                   OUT AppServerSet, OUT UpgradeInfo)
RenewAccessToken(IN AccessToken, IN UserName, IN Password,
                 OUT ReturnCodes, OUT AccessToken, Out RenewalFreq,
                 OUT AppServerSet)
ReleaseAccessToken(IN AccessToken, IN UserName, IN Password,
                   OUT ReturnCodes)
RefreshAppServerSet(IN AccessToken, IN BadQos, IN NoService,
                    OUT ReturnCodes, OUT ServerSet)
```

LSM Functionality Overview

The web pages set up at \\fserv2\home\webpages\pmain.html are intended to provide a framework for discussing overall issues related to subscribe/install/uninstall scenarios, and I look forward to reviewing them with everyone, soliciting feedback, & updating the pages & requirements document appropriately. The material in this section is presented at a much lower level than the web pages.

Run an Application

- ECM invokes **LSMNotifyAppStart** whenever there is a transition from 0 to 1 open files wrt a particular eStream app. ECM can determine the AppID of an

eStream app by inspecting the fully specified eFSD filename; the first level of the directory path is a string version of the AppID. If LSM does not recognize this AppID for some reason, an appropriate status is returned to ECM.

- LSM sends an **AcquireAccessToken** call to via CNI to a SLiMServer [possibly only if the ExpectedExpirationTime for the AppID in question is in the past] and when LSM receives the AccessToken, it places a copy of it in the Subscription Data Set, assigning ExpectedExpirationTime according to the RenewalFrequency indicated by the SLiMServer and setting up a timer alarm to go off shortly before the AccessToken is expected to expire. LSM marks the AppID as ActiveInUse in the Subscription Data Set and returns an appropriate status to ECM.
- Should an AccessToken be denied due to a mandatory upgrade, LSM retries AcquireAccessToken w/ the new RootFileNumber, updates the RootFileNumber in the associated Subscription Data Set entry, & calls **InformUser("Upgrade")**. The RootFileNumber for the specified AppID is always returned to the ECM as an OUT paramater of **LSMNotifyAppStart** so that ECM can take appropriate action if the application version has changed from that being cached.
- Should an AccessToken be denied because it is currently held by another of the user's clients, a copy of the other client's AccessToken is returned. If the user responds affirmatively to **AskUserYesNo("Yank token?")**, then LSM will call **InformUser("Please wait"),** send the token to **ReleaseAccessToken,** & then retry **AcquireAccessToken** at the appropriate time.
- CNI invokes **LSMGetSLiMServerSet** to obtain SliMServers for AccessToken requests. Should CNI have quality of service issues with those servers, it calls **LSMGetNewSLiMServerSet**; LSM uses the **PromptASPNotify** interface to engender the ASP Web Server to send a message containing a SliMServerSet.
- Whenever an AccessToken timer alarm goes off, LSM checks that the associated application is marked ActiveInUse and if so, it issues a SLiMServer request to **RenewAccessToken** and records the returned token in the Subscription Data Set, resetting ExpectedExpirationTime and scheduling another timer alarm.
- ECM invokes **LSMCheckAccessToken** whenever it receives an eFSD request (whether the associated data is currently in ECM's cache or not). If LSM does not have an AccessToken or if ExpectedExpirationTime is in the past, LSM invokes **AcquireAccessToken**. If ECM needs to fetch the requested data from an AppServer, it invokes **LSMGetAccessToken**.
- CNI frequently calls **LSMGetAppServerSet** to obtain a set of AppServers through which to cycle; LSM returns the servers associated with the most recently acquired AccessToken. Should CNI be dissatisfied with the quality of service associated with the AppServers it is given, it calls **LSMGetNewAppServerSet** which sends **RefreshAppServerSet** to a SLiMServer.
- Should an CNI request be denied by an AppServer because the submitted AccessToken was expired, CNI returns an appropriate status to its caller. If the caller was attempting to satisfy an access from an ActiveInUse application, it will undoubtably call **LSMGetAccessToken.**
- ECM invokes **LSMNotifyAppStop** whenever there is a transition from 1 to 0 open files wrt a particular eStream app. LSM sends a **ReleaseAccessToken** call to a SLiMServer [possibly only if the ExpectedExpirationTime for the AppID in

question is not in the past]. LSM marks the application as Inactive in the Subscription Data Set, which indicates that automatic AccessToken renewal should not be continued for this application [LSM may cancel outstanding alarm].

Get/Refresh Subscription Information

- Whenever the CBM receives an **ASPNotify** from an ASPWebServer, it sends an **LSMAspAccountPing** message to the LSM. The LSM checks its ASP Data Set to see if the specified ASP is one that it recognizes. If LSM does not recognize the ASP, it allocates a new ASP Data Set entry and calls **AskUserPassword** to prompt the user for the password associated with this ASP+UserName (asking the user if the password can be saved on the client for future use). If LSM does recognize the ASP, it only calls **AskUserPassword** if the user did not allow the password to be previously recorded.
- Once LSM has the UserName & Password, it issues a **GetSubscriptionList** request to one of the ASP's SLiMServers. For any SubscriptionIDs associated with that ASP that are installed on the client but do not appear in the returned list, the user is informed that he cannot run the app at the current time & is asked if he would like it uninstalled from his client (note that we may be legally required to remove it from the cache, so this may not be a question but rather just a message); **AIMUninstall** is invoked to achieve this.
- For all other SubscriptionIDs, a **GetLatestApplicationInfo** request is sent to a SLiMServer.
- For SubscriptionIDs already in the Subscription Data Set, the RootFileNumber is compared to see if the application has been upgraded. If it has, the user is informed; the new RootFileNumber is recorded for the next run (assuming either that the upgrade is mandatory or that the user indicated he wanted it).
- For SubscriptionIDs that were not already in the Subscription Data Set, a new Subscription Data Set entry is created and **AskUserYesNo("Install App?")** is invoked. If user responds in the affirmative, LSM calls **ECMMount** to notify ECM to prepare to service requests on behalf of this new application & then invokes **AIMInstall,** which can use the eFSD/ECM path to obtain the AppInstallBlock. If user responds in the negative, he is able to indicate "don't ask again" or "ask again later", which is recorded in the Subscription Data Set.
- Whenever the eStream client software starts up or at specific times requested by the user via the ClientUI, **LSMUpdateAllSubscriptionStatus** is called to refresh the eStream client's knowledge of the apps to which the user is subscribed.

Display Subscription Information

- The CUI provides a user interface to display ASP and application info, using **LSMGetAspList, LSMGetAspInfo, LSMGetAppList, LSMGetAppInfo.**

Client Browser Module Overview

The CBM is an eStream client component that runs in association with the client browser. A must-have item is that it be able to communicate a small amount of data in some way to the core eStream client software. A strongly-desirable item is that it be realizable on both Internet Explorer 4+ and Netscape Navigator 4+. A would-be-nice item is that it be able to start the eStream client software if it were not already running.

A number of technologies are currently under investigation. I believe that all three of the items above can be met by native code distributed as a browser plugin, which registers to handle a particular MIME encoded filetype (*.estream) in the HTML stream. Bhaven has created JavaScript that can start Excel & communicate with it, though my understanding of this solution is that it required that the browser be Internet Explorer, that the app be ActiveX, and that the security setting on the browser be at minimum setting.

Issues

- Decide how to store ASP Data Set and Subscription Data Set in nonvolatile memory on the client system.
- Develop mechanism for implementing client browser module.
- Refine needs with respect to Client User Interface.
- Add Prefetcher interface.

eStream App Server Low Level Design

Version 1.2

Sameer Panwar

[redacted]

Functionality

First, some definitions:

eStream page: the smallest unit of data that can be requested by a client from an App Server. Proposed to be 4kB for eStream 1.0.

page set: simply, a sorted list of eStream pages, each identified by a File ID (i.e. AppID & File #) and page # (essentially an offset into the file). This set is restricted only in that all pages in the set must have the same AppID.

client request: a single self-contained message from a client requesting a page set from the server. Each server response to a client request can return a number of pages, and there is a maximum number of pages that the client can request in this message. (TBD, somewhere between 8 and 20 or so).

The primary job of the App Server is to service client requests for application data blocks. The App Server is designed to minimize the amount of CPU time it must consume to satisfy each client request, thereby maximizing scalability. Thus, authentication is performed by a simple expiration time check of an AccessToken provided by the client, and compressed application data is saved persistently.

The App Server serves data derived from eStream Sets. To decouple the performance needs of the App Server from the Builder, we should have a post-processing tool that converts the flat, uncompressed eStream Sets as provided by the Builder into a precompressed format suitable for memory mapping, if the App Server is configured to serve compressed bits. Also, a profiling part of the App Server can be used to monitor for common page sets, and then assemble more optimized replies, which compress the set of pages together as a unit, to take advantage of improved compression ratios. These replies can be stored on disk to save time in rebuilding them each time the server is started up.

The App Server (AS henceforth) views an eStream Set as simply a set of files, and knows no further underlying structure. Thus an eStream Set contains at the start a table (FOST) indexed by File #, and providing the offset into the eStream Set where the associated file data begins, and the size of the file. So the AS just takes the client request of (AppID, File #, Page #, no. of pages), maps AppID to an eStream Set and looks up in the FOST table (File/Offset/Size Table) to find the requested data.

This works slightly differently when the eStream Set file has been pre-compressed by the post-processing tool. The resulting image is the same as before, except now the FOST points to another table, the POST (Page/Offset/Size Table). Because the compressed pages will be of different sizes, this table must by indexed by the Page # to find the relative offset and size of the compressed page data for the file. Thus if an AS is not configured for data compression, the main difference in behavior is that it doesn't do a POST lookup and it doesn't care about coalescing page sequences.

Data type & Data structure definitions

Processed eStream set – this structure is kept on disk and never changes after installation. It looks like:

```
struct {
     ApplicationID appID;     /* for reference, is a 128-bit GUID, see ECM
LLD */
     uint32 maxFileNo;
     boolean compressed_flag;   /* indicates whether the AppFiles are compressed, though maybe we should do it differently? */
     FOST_Entry FOST[<maxFileNo>];
     uint8 appData[<sum of all AppFile sizes, which are variable>] ;
} ProcessedEstreamSet;
```

Since the files in the application are of variable size, we can't make a table out of them, and must indirect out of a table (indexed by the File #) to find their offset location inside the AppData buffer.

```
struct {
     uint32 offset;
     uint32 size;
} FOST_Entry;
```

When the processed eStream set is compressed, then we use the AppFileCompressed structure at the offset indicated by the FOST, otherwise we interpret the data as just AppFile. The AppFileCompressed structure starts with a table that indicates the size and offset of the compressed data that belong to the page it was indexed by.

```
struct {
     uint8 fileData[<size from FOST_entry>]
} AppFile;

struct {
     POST_Entry POST[<number of pages, derived from size from FOST_Entry>] ;
     uint8 fileData[<sum of all FilePage sizes, which are variable>] ;
} AppFileCompressed;

struct {
     uint32 offset;
```

```
    uint32 size;
} POST_Entry;
```

This covers all the structures that live on disk. When we mmap-per-file, that means we make multiple mappings out of a single ProcessedEstreamSet file, at different offsets, one for each file.

Now, for the in-memory data structures (assuming per-file-mmapping):

The primary lookup will be a hash table, hashed on the AppID and FileNo. It should have on the order of 10,000 entries, each table entry containing a list of entries (for collisions). Each list entry contains:

```
struct {
    ApplicationID appID;
    uint32 fileNo;
    uint32 size;   /* size of the mapped Appfile */
    MMap fileMap;
    HTListEntry * next;
} HTListEntry;
```

The Mmap struct just contains any OS-specific-related stuff to manage the mappings, plus a field `char * ptr`, which points to the place in memory that the AppFile (or AppFileCompressed) is mapped. So the hash table looks like:

```
struct {
    HTListEntry * entry[<size of hash table>];
} MMapHT;
```

Hash function is TBD. The hash table should be statically sized large enough to handle the full number of eStream sets up to the maximum memory we will support. Assuming 32 bytes being used per entry, that implies about 1 MB to handle 30k files, which is no problem. (Maybe we should reserve entries for 100k files or more?)

<u>Configuration</u>: Each AS must obtain configuration data, either directly from the database or from the monitor in its startup message. The required data is (with the config param names and datatypes):

```
AppList          vector of ApplicationID's (128-bit GUIDs)
ServerPort       uint16
MonitorPort      uint16
SLiMKey          uint (size TBD, depends on actual algorithm)
ClientTimeOut    uint32
CompressionFlag uint32
```

<u>Network communication</u>: The AS talks only to clients and the server monitor via the network. The server monitor communication will be described as part of the monitor heartbeat protocol. The AS-client communication will be described in a separate docu-

ment. The AS will time-out and close connections that have been idle for some amount of time (a few seconds).

[maybe combine multiple responses into a single send socket call (will only work for TCP probably, since proxies won't like multiple server responses)?]

Interface definitions

The AS is optimized to do one thing only: serve pages from the read-only file system part of eStream, so there is just one interface with the client. Anything the client can care about in an eStream set is just another file to the AS, including the AppInstallBlock, and directories/metadata. The AS only returns the data the client requested, nothing extra.

```
struct {
     uint32 fileNo;
     uint32 pageNo;
} PageRequest;

struct {
     uint32 errorCode;
     uint32 compressedFlag;
     uint32 fileNo;
     uint32 pageNo;
     uint32 offset; /* offset into pageData below */
     uint32 dataSize;
} PageReply;
```

PageReadRequest

Caller:	Client	
Callee:	AppServer	
Input:	uint32	appId;
	eStreamAccessToken	accessToken;
	uint32	numPagesRequested;
	PageRequest	pageSet[(numPagesRequested)];
Output:	uint32	numPagesRequested;
	PageReply	pageSetReply[(numPagesRequested)];
	uint8	pageData[(sum of all page data)];
	uint32	globalErrorCode;
Global Errors:	INVALID_ACCESS_TOKEN	
	EXPIRED_ACCESS_TOKEN	

INVALID_APP_ID
EXCEEDED_MAX_REQUESTABLE_PAGES

Errors within
PageReply: INVALID_FILE_NO
INVALID_PAGE_NO
SERVER_ERROR (probably should be logged, and should cause an alert if too many occur in some time period, including errors that don't get returned to the client.)

AppServers don't ever talk to the database (it would be a waste of licenses considering the number of AppServers we'd have and their infrequent accesses). Instead, they obtain all their relevant control information from the server monitor.

The exact interfaces are TBD, but from the monitor they will provide configuration information, AppServer state change requests, and add/remove requests to the list of apps being served. Going back from the AppServer to the monitor, it will report load (average response time) on a per app basis, and server state, along with the heartbeat.

Component design

Interesting issues to deal with:

Scalability/Performance

Since scalability (and thus performance) is critical for the AS, let's go over how CPU and memory are used.

Memory

Performance is maximized when virtually all client requests can be satisfied by retrieving the desired pages from RAM, because RAM is far faster than disk. Thus the amount of RAM available will put an upper bound on the number of apps that a single AS can serve efficiently. Since server RAM won't grow as fast as the total size of all apps available as eStream sets, this means we'll have to heterogenize servers, where each server specializes in a subset of apps, limited by available RAM. For eStream 1.0, this component of AS configuration will be handled manually, the eStream administrator assigning apps to servers. In the future, the set of App Servers should automatically reassign apps dynamically to balance load.

But this is just one level of memory, committing RAM to a set of apps. There still remains the question of how to best utilize that RAM for each app, since some files are used far more often than others. This immediately means that for efficiency we must overcommit RAM, because if we allocate an entire eStream set into RAM, we're using precious resource to hold data that may be requested only very rarely. Instead of having to manage our physical RAM manually to accomplish this (such as with a cache), an easier approach would be to take advantage of virtual memory (VM) to automatically keep

the hot pages in RAM, with the remainder available (again **automatically**) off disk (via memory mapping the eStream sets). That way the server can satisfy any possible client request for any app it serves, but is optimized to be the most efficient over all clients. But this only works if enough VM is available. (Time for some back-of-the-envelope numbers.) Given that an app seems to have something like only 20% of it being hot (from our current limited data from the prototype), this means VM must be at least 5x of RAM for maximum efficiency. Given that a process has about 2 GB addressable VM, this corresponds to about 400 MB of RAM. Beyond that size (which is not uncommon), we don't have enough VM to efficiently overcommit our memory (by mmaping entire eStream sets). So now our choice is to either manually manage a memory cache (and all the attendant coding, bugs, etc.), or to mmap at a finer granularity.

Note that the effective virtual memory required by an app is increased when compression is used, to handle the extra compressed page sets. They'll probably double or triple the RAM footprint by hot pages (due to redundancy), but only increase the overall VM footprint by 1.2 – 1.5. The consequence of this is that the overcommit ratio goes down to 1.5 / (3 * .2) = 2.5, though the amount of apps servable is reduced to 1/3 (!!). Now 2 GB virtual address space corresponds to 800 MB of RAM. This means we should be able to just memory map entire eStream sets, up to 2 GB worth, and be confident we're utilizing RAM efficiently, assuming the server has about 800 MB of RAM. A server with less RAM will likely thrash, and those with more will likely see little improvement in the number of apps they can serve via memory mapping.

A loss of 2/3 in the number of apps an AS can serve I think is too great a sacrifice, too great a loss in app scalability (need 3x the number of servers as before!) for what is about a 15-30% greater effective bandwidth at the client. The root of this problem is the redundancy (costly in physical memory), because the compressed page sets will contain the same page in multiple sets. This is similar to the redundancy that appears in trace processors and dynamic translation, which places extra memory demands in both those cases. I think we must completely eliminate this redundancy to achieve the goals we desire, either by (1) not using compressed page sets, and just sending multiple individually compressed pages, or (2) ensuring a page appears in only one compressed page set. [There further potential loss of effective memory size when using compressed page sets since they'll be allocated in 4k chunks, thus wasting about 2k on average; we'd have to batch them up together in files to minimize this… Also, saving the compressed page sets to disk introduces extra complexity to the AS because we'd have to properly handle recovery (i.e. what if the system crashes while we're writing the sets, which if we're memory mapping is totally out of our control). Because of this robustness requirement, and the fact we need to be 100% sure we're serving good bits (lest we crash a bunch of clients), this needs to be thought out very carefully if we want to do this. My opinion is that we should defer implementing compressed page sets until we better understand the tradeoffs, and good profiling schemes. In particular, will the AS be mostly bandwidth-limited, memory-limited or CPU-limited?]

Separately from this, we should consider the effect of per-file memory mapping (ignore the compressed page sets now). This has the impact of requiring many more mmap's

from the OS, but promises better use of the limited virtual address space. In this scheme, we mmap each file into VM as it is referenced by a client. If only hot files are referenced, then the RAM footprint is the same as before, but VM is only used for the hot files, not the entire app, probably about 30-50% greater in size. Thus the overcommit ratio then becomes 1.5, much better than the 5 with full app mmapping. So 2 GB of VM corresponds to 1.3 GB of RAM, much better than the 400 MB with full app mmapping. However, this assumes that VM is used in a cache-like manner, evicting not recently used mmap's, since as uncommon files are referenced, they eat up more and more VM. Once VM is totally used up, then replacement policies and eviction (and fragmentation of virtual address space) become issues, just as with a manually managed cache. One solution is to simply purge all mmap's and start from scratch, which is simple and reliable, especially considering the AS is multithreaded (if this is done, the above analysis doesn't hold, and performance becomes a function of how often VM is cleared). Another possibility might be to use the profiling mechanism and only place sufficiently popular files in mmaps and do regular file system accesses for the rest.

Of course, the alternate option for managing physical memory is to know its size, and manage a cache manually. One advantage here is that the AS would know the physical memory consumption and usage (unlike when the OS was handling everything), which may help with load balancing. The main advantage is that there are no artificial limits (overcommit ratio is irrelevant), and only physical memory size is the true limit, and this approach can map any number of eStream sets (with any size of files) to any amount of physical memory up to the virtual address size (4 GB). Then memory management becomes an issue (what do you do once all your RAM is full), which can be painful in a multithreaded environment. Again, we can just invalidate the whole cache as an option, but this will probably happen more often than with the per-file-mmapping case, unless RAM is greater than the maximum that the per-file-mmaping approach can handle. If the wholesale cleanup approach is used, then allocating fixed size chunks may not be needed, and we could potentially get better memory usage by packing compressed pages more tightly (e.g. 16-byte aligned vs. 4kB aligned), which is another potential advantage. Maybe instead of wholesale cleanup, we mark the most commonly used pages, and then just compact those and dump the rest (say 50%). The main issue with this approach is potential redundancy with respect to the OS disk cache (which is shared in the mmap approach), and assumption that our caching policies will be better than the OS's. Also, lookups get messier, since we need a bigger lookup table to index via page # as well.

Yet another option is to use multiple processes instead of multiple threads, one process per app being served, thereby releasing us from the 2 GB VM limitation. However, this introduces the issue of multiplexing requests from the network via IPC, and more load on the server monitor. On x86 NT, a Very Large Memory feature is available that can provide 36-bit addressing per process; we may want to use this even though it won't be available on regular Unixes (and probably not x86-linux).

In summary: per-eStream-set-mmapping is probably too wasteful of virtual address space. Per-file-mmapping is much better, but then memory management becomes an issue, suggesting a simple throw-away-and-start-over solution. However, given that solu-

tion, if a lot of physical memory is desired, a manual cache approach may be better (the better packing should overcome any loss due to redundancy with the OS disk cache). Compression of page sets invokes several issues that probably can't be fully addressed until after 1.0. **Bottom line: the target now for 1.0 is to use per-file-mmapping with per-page compression (but no compression of page sets).** Also, we should instrument the system to allow us to easily collect the relevant data (mem usage, CPU load of different routines, etc.) to help guide us in further evolution of the system to improve performance (e.g. compressed page sets or explicit page cache).

CPU

The main work of the CPU is as follows (encryption is assumed to be done by hardware since its CPU impact is severe):

1. OS system call to retrieve request from network.
2. Decode client request.
3. Validate AccessToken.
4. Lookup AppID, File # in primary lookup hash table. (If mmapping eStream sets, instead lookup in App table, then lookup in FOST).
5. If mmapping then (if uncompressed, no further lookup, if compressed, then lookup in POST to find page and size), if explicit cache then look in B-tree (secondary lookup).
6. If lookup fails, then bring in the data off disk (either mmap or file system call).
7. Copy page data to reply buffer.
8. OS system call to send reply to network.

However, if compressed page sets are used, lookups get more complicated, with a different set of tables to check for an appropriate page set first (and lookup failures incur potential decompression/compression). It appears the least amount of CPU time is probably incurred when doing per-file-mmapping. All pages held in memory are kept in compressed form to save repeated compression of the same data, so pretty much all the work is in lookups and memory copies. Potentially the AccessToken validation will use hardware assist. Lookup failures (i.e. having to go to disk) should be relatively uncommon, and memory should be sized to ensure that.

However, since the AS will run in user mode, this incurs the penalty of two extra copies (from the network buffers) and switching between kernel and user mode twice. If this is enough of a problem, we'll have to consider implementing the AS to run in the kernel (all commercial NFS, etc. implementations run in the kernel), which means we should choose our implementation to be compatible with that approach. In particular, we may not be able to rely on the virtual address space not being fragmented, so mmapping full eStream sets may be impossible. Plus robustness of the server becomes even more important, and portability issues arise. For the 1.0 release, we plan to implement the AS in user mode keeping the possibility of moving to kernel mode in the future, and will collect data from 1.0 (or derived prototype) to evaluate the actual benefits.

Disk: Since we are relying heavily on the common pages being in memory, we could possibly even consider storing the processed sets on a network disk, i.e. remote from the app server itself. However, such sharing won't work well for compressed page sets since

they are written to at runtime—it would be extremely messy to handle dozens of app servers trying to add many compressed page sets (possibly the same) to a set of shared files.

Multithreading model

The approach will be to have a single boss thread which pulls things out of the network port and stuffs client requests into a queue and a bunch of worker threads which grab requests and send back the replies. Simple enough, but this raises the issue of thread control, since the boss also needs to be able to handle threads that die or hang and kill and restart them. The boss thread will monitor the worker threads and provide load/hearbeat info to the monitor through the server manager thread, thus giving visibility to the server monitor of the health of all the worker threads.

Load balancing

To be described elsewhere? (appears in SLiM server LLD)

Security

There are two levels of security involved in the AS. First, we must prevent clients who don't hold valid licenses from gaining access to the licensed binaries. This is accomplished by the client obtaining an AccessToken from the SLiM server and presenting it to the AS upon every request. The AS can then use the SLiM server's public key to test the authenticity of the AccessToken (to protect against forgeries), and then can test the authentic expiration time of the AccessToken. Second, we must encrypt the actual data being sent on the wire to prevent third parties from gathering the binary data covered by the license. Since the data coming out is somewhat obfuscated anyway (files are identified by arbitrary IDs, with our own strange message formats and compression and all in random pieces, etc.) it is not clear how much extra protection is really necessary, i.e. what do the license issuers actually want? We should use a common scheme like SSL to perform this encryption. It has been decided that the encryption load for this would be too great, and thus the data send back will be unencrypted. We may use SSL for authentication purposes only (i.e. null-cipher), if that is cheap enough.

Also, a possible optimization for checking AccessTokens would be to cache recently used AccessTokens along with a signature/hash. If a token presented by a client matches, then we can skip the authentication step (since we've done it once already) and just check the expiration time.

Robustness

The AS must be very robust. It must catch OS call errors and handle/log them as appropriate, and deal with threads that hang or die. Thus it needs to aggressively check for error conditions and possible failure modes. The AS also needs to track relevant resources (e.g. sockets, memory) and carefully manage/reclaim them so as not to exceed any limits or to degrade performance. And of course, the AS needs to check all data coming in from the client, to deal with ill-formed requests, and illegal values (e.g. huge negative indexes, etc.), and perform no potentially dangerous operation without validating parameters. This becomes even more important when we eventually move the AS to run in kernel mode. The AS also needs to be as stateless as possible, to minimize recovery time, and if it does perform writes to disk (such as for the compressed page sets), do so in a reliable fashion conducive to quick recovery. Any unreliability in the AppServer will negate any benefit of scalability we have over our competitors.

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

The various components of the AS are not too large or complicated: The request dispatcher (to worker threads), the hash table, the compression code, the AccessToken checking code, etc. These shouldn't be too hard to do reasonable testing on in isolation.

For the post-processor component, we'll have to build some sample Estream Sets as input, but it'll be hard to tell whether the output is correct without having a minimal working AS.

Cross-component testing plans

The best approach will be to perform incremental implementation and testing. I.e. we build the core functionality that is required (i.e. can start with just regular i/o reads), and then add the more performance-related stuff later (adding mmaping, and then the hash table & AccessToken checks), while testing the entire system as pieces are gradually added (of course performing sanity-check and other minimal testing on the pieces first if possible). Compression can be added last.

To actually drive the AS, we'll need a test client, which will be designed to just shoot off a series of read requests to the server. The file data returned could then be written to files, and this can be compared against the original set of files used to create the Estream Set we started with, to check that the data was received properly. For checking error conditions, a log of errors can be written and compared against a reference log for those requests we expect to fail.

Stress testing plans

To accomplish this, we should run multiple independent test clients (on the same machine and on different machines), and increase the frequency of requests (to stress the AS's threads and sychronization, and communication routines), and the number and size of files referenced (to stress the hash table and memory). Each test client can then check whether the data and errors it got back were as expected, like in the above subsection.

Coverage testing plans

Should we use some kind of code coverage tool for this?

Performance testing plans

Since performance is critical, we should take the time to evaluate the AS's performance characteristics. We need to crank up our stress testing until either bandwidth or CPU saturates, and record the request rate that generated it. We should compare how this point responds to high numbers of clients with fewer requests per client vs. fewer clients with higher requests per client. We'll need to profile the system to find bottlenecks to tweak more performance out of it, and learn how well our original design assumptions hold up. Depending on whether CPU or bandwidth (or memory) saturates first, we may want to modify the system's tradeoffs to improve scalability further, and otherwise note which components a customer should upgrade for better performance. Also, if we think we can come up with reasonable client access pattern profiles, we may want to use those to estimate the actual number of real-world clients an AS can support. As part of this, we'll probably want to run the AS in-house once it is mature enough (eat our own dogfood), and then farm out app upgrades, etc. (play out some of our scenarios) and see what happens to the AS's (do they choke or what).

Availability testing plans

We will also need to test our failover and load balancing capabilities. This will require several test machines with the monitor in place to start and stop servers, and have clients be aware of multiple AS's and respond appropriately when an AS stops responding. For load balancing, we'll probably want a bunch of test clients with a variety of access patterns and see how well their requests are distributed.

Upgrading/Supportability/Deployment design

App Servers will possibly need to version their interface with clients (requiring clients to state the version they're expecting), but will also need to support older versions. We may also modify the Estream Set format (or just the processed set format), but that should be handled by upgrading both the AS and post processor and then regenerating the processed sets.

For supportability & deployment, the AS will report error conditions and load to the server monitor, which is used by the customer.

Open Issues

1. Is there a limit to the # of possible mmaps?
2. Is there a single system call to unmap all mmaps?

eStream Server Component Framework
Low Level Design

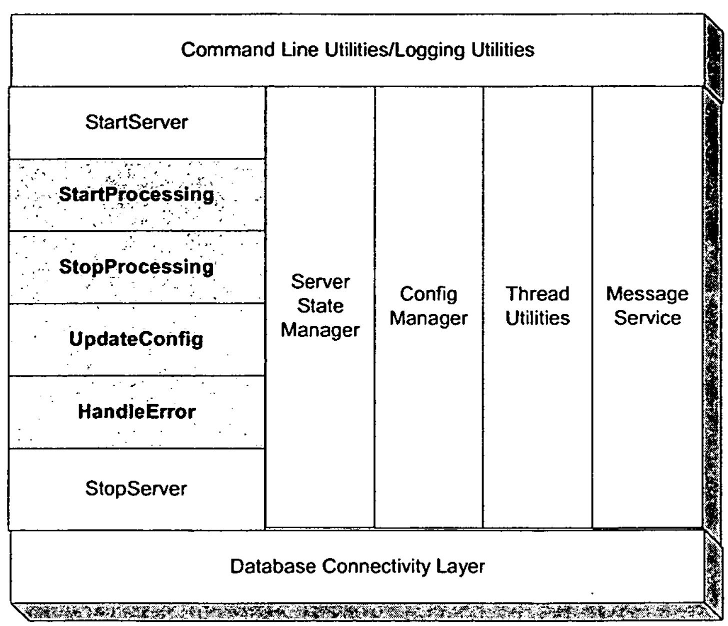
Michael Beckmann
[redacted]

Functionality

The *Server Component Framework* provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. The component framework ties together many of the core utilities provided for the server components.

The advantage of the framework is that heterogeneous server components can be managed in a consistent manner with the expectation that all server components will communicate and behave consistently within the system.

All server components with the exception of the web server will be built on top the *Server Component Framework.* To make use of the *Server Component Framework,* a specialized server component will need to extend the framework by implementing the methods high-lighted in gray. Implementing these interfaces makes the specialized server component "plug-able" within the framework.



The following table give a brief description of each of the routines that need to be specialized by each server component to make it plug-able into the Server Framework:

StartProcessing	Specialized server component routine to request the server component to start processing work.
StopProcessing	Specialized routine to request the server component stop processing work and transition into an idle state
UpdateConfig	Specialized routine to dynamically update configurations while a component is either in the processing or idle state.
HandleError	Specialized routine to handle the occurrence of an error

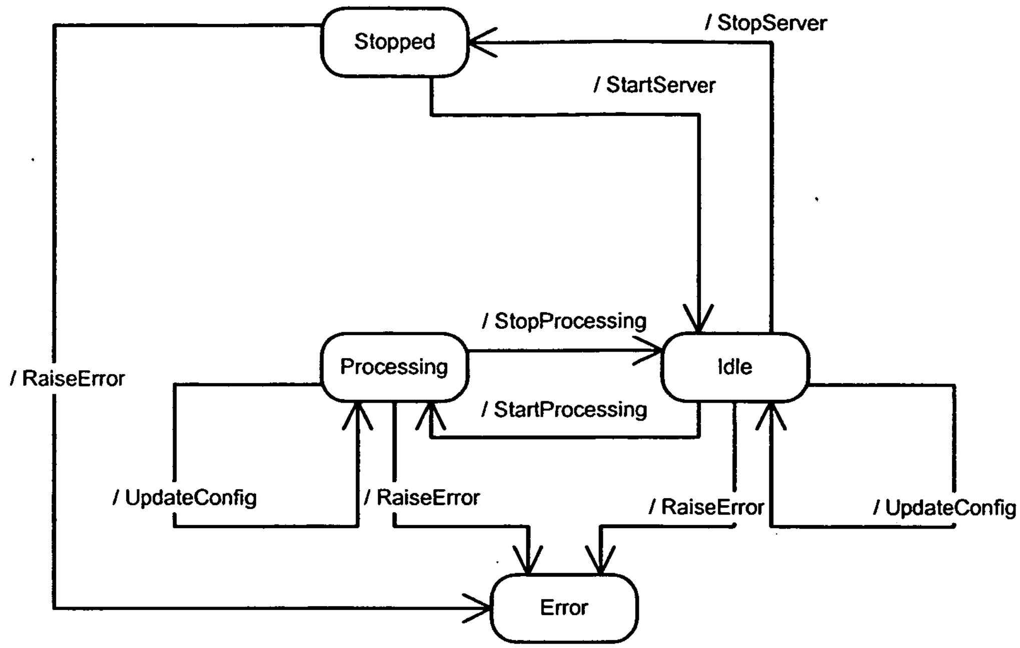
Server State Manager:

At the heart of the server component framework is the *Server State Manager*. The server state manager is a set of interfaces that initiate and manage state changes within a server component. All Server components, by virtue of being built on top of the component framework, can be managed uniformly across a deployment.

The *Server State Manager* implements a simple state machine that is shared between components. It manages the state transitions within the server component. Additionally, the state manager maintains current state information for each server component and logs state transition history in the event that a server component terminates unexpectedly.

As specified above, each server component is required to implement a number of transition methods, with pre-defined signatures, which the state manager will execute when making a state transition.

The following diagram shows the state diagram and the associated transitions:



Message Service:

The *Server Component Framework* depends on a message service which is used by the Server State Manager and Configuration Manager to communicate with the System Monitor.

The *Server State Manager* uses the messaging service to listen for state change requests from the System Monitor which it satisfies by returning the current state, any up-to-date status, and load information.

The *Configuration Manager* uses the message service to request configuration information from the *System Monitor*. Although each server component could easily go to the database for configuration information, it has been decided to go through the monitor as to save db licensing costs.

See below for more details on messaging protocols for the *Server State Manager* and the *Configuration Manager*. Also, refer to the low-level design document for details on the design of the eStream Messaging Service (EMS).

Configuration Management:

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database (indirectly).

- Servers can load the configuration for a given name.
- Servers can load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:

port 8080

An example of nested name values would be:

Applications:
word.exe *windows2000sp3*
excel.exe *win98sp4*

On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:

Applications word.exe windows2000sp3
Applications excel.exe win98sp4

A common set of configurable parameters is defined for all server components. These configurations are maintained by the *Server Component Framework* in collaboration with the *Configuration Manager.* All configuration information is persistently stored within the database. The common configurations are used to initialize the server component after the component process has been launched. Refer to the configuration table below for more details on common configurations. Specialized server components can support additional configurations (non-common) depending on the server type. These configurations are read from the database and updated when a server component starts processing. They can also be updated dynamically while a server component is processing through the use of the **UpdateConfig** interface.

The list of common configurations include:

Information	**Supports Dynamic Config**	**State**	**Description**
ServerID	No		Unique identifier for server components. This server identify is unique within a deployment. This ServerID is not known to eStream clients. Its purpose is as a handle to uniquely identify server components.
ServerType	No		Identifies the type of server component. One of the following applies: ▪ Primary Monitor

			▪ Backup Monitor ▪ Application Server ▪ SLiM Server
DbUser	No		User name string required for database connectivity for this server ID
DbPasswd	No		Database password associated with the DbUser
Dsn	No		Data Source Name used to access the database.
PortNum	No		PortNumber used for light-weight messaging listener
MachineID	No		Machine ID is used to get at important machine information needed for all server components such as: ▪ IP address for the machine server component is hosted on ▪ Domain name for the machine ▪ Machines name
AutoReStart	Yes	Any State	Flag indicating that server component process can be restarted automatically without manual intervention. This info is consumed by the System Monitor.
HeartBeat-TimeOut	Yes	Any State	Specifies the timeout period for the listener. If the timeout period is reached. The component assumes that it has lost the connection. All Server components have a listener by which they receive instructions from the primary system monitor. Even the monitor has a listener that communicates with the Server Admin UI.
HeartBeatRate	Yes	Any State	Frequency at which the heart-beat is sent to this server component. Specified in milliseconds. This item is consumed by the System Monitor.

Command Line Utilities:

The *Command Line Utilities* component provides a consistent way to define and process command line arguments. To use this utility, the using component must define a table of arguments, which defines the valid set of arguments, whether or not they are required, and any default values.

Arguments are specified on the command line as name/value pairs. The utility implements the following command line syntax to support the name/value pairs. The argument syntax is defined as follows:

<name>=<value>

name	Name is an alpha-numeric identifier. The Name can be of arbitrary length as supported by the system however shorter names are recommended. Names are case sensitive
value	Any alpha-numeric value. Punctuation characters may also be used. Values are

	case sensitive

There can be no spaces between the <name>, "=", and the <value> elements. The existence of one or more spaces or tabs delineates separation between arguments on the command line.

Example: server.exe sid=267 dsn=oracle user=michaelb passwd=mypasswd

- If a named argument is specified more than once on the command line, subsequent arguments will cause a diagnostic to be issued and the argument will be ignored.
- This utility allows the user to specify default values for arguments. If a default value is defined then the argument will be processed with its default in the event that the argument is not specified on the command line.
- This utility allows the user to tag specific arguments as required. If the required argument is not specified on the command line this utility will raise a diagnostic for the required argument. Not specifying a required argument will cause a fatal error.

The following options are supported:

sid	Server Component Identifier. Each server component within a deployment is uniquely identified via the sid. The sid is a handle into the database for accessing information unique to a specific server component.
dsn	Data Source Name. A data source name is necessary to establish an ODBC connection. Data Source Names are generated by an ODBC administrative tool
dbuser	User name. For database access security, all components need to connect as a specific user.
dbpasswd	password associate with the dbuser

Logging Utilities:

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
    - 4-Low : A warning which can be ignored.
    - 3-Medium: A warning which needs to be looked into.
    - 2-High: Recoverable Error in the component.
    - 1-Critical: Fatal Error. Needs admin assistance.

- Logging level should be configurable. The following levels are to be supported.
  - 0: Only errors will be logged. This will be the default level.
  - 1: Errors and Warnings to be logged.
  - 2: Errors, Warnings and Debugging information to be logged.
  - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.
- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
  - Any existing <logfile>.bak will be deleted from the system.
  - The current <logfile> will be backed to <logfile>.bak
  - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]
[TimeStamp] [Thread ID] [Priority] [Message]
 ...
[FOOTER]

An example of this log format would be:

```
*********************************************
Omnishift eStream Application Server
Server Started.
StartTime: [redacted]:31:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
*********************************************

[[redacted]:16:31:19 -0700] 0 2-High Cannot connect to the database. Invalid Username/Password.
[[redacted]:16:31:19 -0700] 1 1-Critical Cannot start the HTTP listener at port 80.
[14/Aug/2000:16:31:19 -0700] 0 1-Critical Shutting down the server.

*********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime: [redacted] 16:35:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
*********************************************
```

Format of Access Log Message:

[HEADER]
[TimeStamp] [Thread ID] [Message]
[FOOTER]

Data type definitions

Server State:

The server components can be in any one of the following states:

State	**Description**
STOPPED	If a server is in the STOPPED state then the component process is not running.
IDLE	Server component is up and running. The server has been initialized with the common configuration and the messaging system has been enabled. The listener is actively waiting on the System Monitor for transition requests. The server component is not processing any work specific to this servers specialization.
PROCESSING	Server component is actively taking requests and processing work specific to its specialization. ie. serving access tokens, and application file requests.
ERROR	An error has occurred in the system. Unless the server component is configured with **AutoReStart** and ERROR state must be manually cleared by the server-side administrator.

Server State Transitions:

Changes in server component state are initiated either by the *System Monitor* or directly by the server-side administrator for the system monitor. The exception to this is when an error condition is raised by a server component. In this case, the component will initiate the state change itself. The following state transitions are supported:

Action	**Description**
START_SERVER	Server is expected to be in the STOPPED state. If a server component is configured to support AutoReStart then the ERROR state is also a valid state from which to initiate this action.
STOP_SERVER	Causes the server to exit its process. The server can be stopped from any state.

START_PROCESSING	Causes the server to change from the IDLE state to the PROCESSING state.
STOP_PROCESSING	Causes the server to change from processing to IDLE state.
UPDATE_CONFIG	Request that the server read its configuration from the configuration manager and change its configuration.
RAISE_ERROR	Request that the server go to ERROR state. This causes an error handler to be called. If the error is fatal it will cause immediate termination of the server process.

Finite State Table:

```
FSMTableEntry ServerStateMgr::FSMTable[] =
{
	{ START,{{START_SERVER,STOPPED, START_SERVER,NULL},
		{START_PROCESSING, STOPPED ,START_PROCESSING, NULL},
		{NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

	{STOPPED, {{START_SERVER, IDLE, NULL_REQUEST, &StartServer},
		{START_PROCESSING, IDLE, START_PROCESSING, &StartServer},
		{RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
		{NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

	{IDLE,{{START_PROCESSING, PROCESSING,  NULL_REQUEST,
&StartProcessing},
		{STOP_SERVER, STOPPED, NULL_REQUEST, &StopServer},
		{RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
		{UPDATE_CONFIG,  IDLE, NULL_REQUEST, &UpdateConfig},
		{NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

	{PROCESSING,{{STOP_PROCESSING, IDLE, NULL_REQUEST,
&StopProcessing},
		{UPDATE_CONFIG,PROCESSING, NULL_REQUEST,
&UpdateConfig},
		{STOP_SERVER, IDLE, STOP_SERVER,  &StopProcessing},
		{RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
		{NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

	{ ERROR,{{STOP_SERVER, STOPPED,NULL_REQUEST, NULL},
		{NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} }

	{NULL_STATE,{{NULL_REQUEST, NULL_STATE, NULL_REQUEST,
	NULL} } }
};
```

Messaging Service Protocol:

A light-weight messaging protocol is needed to facilitate communication between server components. The primary purpose of the messaging protocol is to communicate transition requests to the server components. In response, server components communicate state, status, and load information back to the *System Monitor*.

The messaging protocol supports two primary message types. 1) Requests for the *System Monitor* to perform on other servers. 2) Requests to the server components themselves. These message types are distinguished through the protocol as described below. If the receiver ID and the target ID are identical then the request is for the receiver. If the target is different than the receiver, the message is for the *System Monitor* to enact a request on the target component.

All requests are required to be acknowledged. Without an acknowledgement the message is considered un-received.

OpCode	**senderID**	**receiverID**	**targetID**	**Data**

The following table describes the protocol used by the Server State Manager in its communication with the System Monitor.

OpCode	**Description**	**Data**
0x01	Request for current state	None
0x02	Acknowledgment	▪ Current state ▪ Load info ▪ Status info
0x03	Stop Server request. Acknowledged with 0x02 message	None
0x04	Start Server request. Only valid for System Monitor. Acknowledged with 0x02	None
0x05	Start Processing Request. Acknowledged with 0x02	None
0x06	Stop Processing Request. Acknowledged with 0x02	None
0x07	Update Configuration Request. This is a request for a server component to request its specialized configuration information from the System Monitor and update itself. Acknowledged with 0x02.	None

Interface definitions

Server State Manager:

```
class ServerStateMgr
{
```

```
private:
        ServerState CurrentState;
        static FSMTableEntry FSMTable[];

public:
        ServerStateMgr(void);
        ~ServerStateMgr(void);

        ServerState SetState(ServerState);
        ServerState GetState(void);
        ServerState ProcessRequest(ServerRequest);
};
```

SetState	**Description:** Sets the current state of the server component. 1. Log the state change request 2. Update the state field within the server component in memory data structures. 3. Send message to requester informing them of the successful state change. Note: **SetState** does not update the database directly as in the orginal design. The database is updated by the *System Monitor* once it has received an acknowledgement. A state transition is not complete until **SetState** returns successfully and the Monitor has update the database. **Input:** state value to set current state to. **Output:** current state after the new value has been set. If an error occurs will go to error state. **Errors:** 1. Invalid state argument 2. Failure to either connect or commit state change to the database.

GetState	**Description:** returns the current state. This function does not read from the database to get the current state. The assumption is that if the server component is up and running and that it maintains a valid state. **Input:** none. **Output:** returns the current state. **Errors:** None. Will always return a valid state.

ProcessRequest	**Description:** request to the Server State Manager to change server state. This routine implements the guts of the state machine. 1. Get the current state, and transition request 2. Index into the FSM table and continue to transition from state to state until the transition request is satisfied. 3. Each state transition calls the specialized transition routines for each component. 4. Call to **SetState** to complete each state transition. 5. In the case of an error the state machine will process a RAISE_ERROR request which will call the specialized **Han-**

	dleError and transition to the ERROR state. **Input:** server transition request. Refer to table of valid requests defined above. **Output:** current state after the request has been completed. **Errors:**

Server Component Framework:

```
class ServerComponent: ServerStateMgr{     // abstract base class
private:
        ErrorInfo*      Error; // maintains error if error was detected
        ServerConfig*   Config;  // holds common configuration
        Connection*     Listener; // messaging        utility
public:
        virtual int StartServer(void); // may be specialized by a server component
        virtual int StopServer(void); // may be specialized
        virtual int StartProcessing(void) = 0; // must be specialized
        virtual int StopProcessing(void) = 0; // must be specialized
        virtual int UpdateConfig(void) = 0; // must be specialized
        virtual int HandleError(void) = 0; // must be specialized
        void Run(Request);
}
```

StartServer	**Description:** Called by the *Server State Manager* when a server component is to be started. The **StartServer** routine is provided as part of the SeverComponent class. It performs the following: 1. Send request to System Monitor to request an update of common configuration information. 2. Apply the configuration information to the server component. 3. Construct a listener connection object and start the message service. 4. Return success or failure. Note: ▪ This routine must return immediately to the main thread. Otherwise the *Server State Manager* will be blocked. ▪ Successful return from the **StartServer** routine will put the server into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error condition

StopServer	**Description:** Called by the *Server State Manager*. 1. Perform any necessary cleanup. 2. Send last acknowledgment confirming shutdown to requester 3. Shut down the messaging system and the listener. 4. exit process Note: The monitor will update the database and perform logging.

	Input: None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error value

StartProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by each specialized server component. This routine is used to provide all functionality unique to different types of servers. 1. Spawn a primary processing thread (also known as the boss thread). a. Read server specific configurations unique to this type of server component from the System Monitor b. Spawn worker threads. Depending on the server type this routine does the heavy lifting to either process access tokens and renewals in the case of SLiM server, or process file requests for application servers, or manage and monitor the server components in the case of the *System Monitor*. Note: ▪ This routine must return immediately so that the *Server State Manager* can continue to operate in the main thread. ▪ This routine may make use of the *Server Configuration Manager* for obtaining specialized configuration information **Input:** None **Output:** Value of 0 if successful else error condition. **Errors:** TBD

StopProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by the specialized server component type. 1. Reverse all actions performed by the **StartProcessing** routine. All worker threads should be joined or pooled in waiting state. Successful return from this routine will put the server component into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

UpdateConfig	**Description:** Called by the *Server State Manager*. This routine must be defined by the specific server component type. The purpose of this routine is apply dynamic configurations or update specialized configurations that are unique to this server component. <may require adding a new state to separate dynamic and static configurations> **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

HandleError	<u>**Description:**</u> Component defined error handling routine to handle errors such as timeouts, etc. This routine will need to handle a number of error cases as are possible by the specialized component. The error information is maintained with the ServerComponent class. <u>**Input:**</u> None. <u>**Output:**</u> Integer value designating a handled error or failure. If the error cannot be handled then it is fatal. <u>**Errors:**</u> TBD

Run	<u>**Description:**</u> This routine implements the main processing loop for the server component and runs in the main thread. This routine drives the server component by initiating state requests from the *System Monitor*. Note: The *Server State Manager* always runs in the main thread. 1. Call **ProcessRequest** to transition the server component into the initially requested state. 2. Enter main processing loop a. Check for requests from the message service. b. Call **ProcessRequest** to service the request. c. Send acknowledgement for the request to the message service. Acknowledgement includes new state, load info, and status. <u>**Input:**</u> Initial Transition Request <u>**Output:**</u> None. This routine should never return <u>**Errors:**</u> None.

Server Component Main Loop:

The following main loop is common to all server components:

```
void ServerComponent::Run(ServerRequest Request)
{
        ProcessRequest(Request);
        while (1)
        {
                Request = Listener->GetRequest();
                ProcessRequest(Request);
                Listener->AckRequest(Request, GetState, GetLoad, GetStatus);

        }
}
```

```
#include "ServerArgs.h"
#include "Server.h"
```

```
int main(int argc, char* argv[]) {
        Args = new ArgList();
        Args->ProcessArgList(argv, argc);
        Server = new ServerComponent(GetValue(SID),
                                     GetValue(DSN),
                                     GetValue(DBUSER),
                                     GetValue(PASSWD));
        Server->Run(START_PROCESSING);
}
```

Command Line Utilities:

```
class NameValuePair
{
        private:
                char* Name;
                char* Value;
        public:
                NameValuePair();
                ~NameValuePair();
                char* GetValue(void);
                char* GetName(void);
                char* SetName(char*);
                char*SetValue(char*);
};
```

```
typedef int (*pFunc)(NameValuePair*);

struct ArgTblEntry
{
        char*   Name;
        bool    Required;
        char*   DefaultValue;
        pFunc   ProcessFunction;
};
```

```
ArgTblEntry const ServerArgsTbl[] = {
        {"sid",         true,   0,      &ProcessSId},
        {"dsn",         true,   0,      &ProcessDsn},
        {"dbuser",      true,   0,      &ProcessDbUser},
        {"dbpasswd",    true,   0,      &ProcessDbPasswd},
        {0,             0,      0,      0}
};
```

```
typedef vector<NameValuePair*>    ArgVector;
```

```
class ArgList
{
	private:
		ArgVector			ArgVec;
		const ArgTblEntry*		ArgTbl;

	private:
		NameValuePair*			ParseArg(char* Arg);
		char*				ParseName(char* Arg);
		char*				ParseValue(char* Arg);
		int				ProcessArg(NameValuePair*);
		int				FinalizeArgs(void);

	public:
						ArgList(const ArgTblEntry*);
		int				ProcessArgList(char* argv[], int argc);
};
```

ProcessArgList	**Description:** Process the entire argument list. In a loop for each argument argv[] … 1. Call **ParseArg** passing in argv[]. 2. **ParseArg** passes the result to **ProcessArg** 3. After processing the entire argument list and exiting the loop call **FinalizeArgs** **Input:** argv and argc as passed into main() entry point **Output:** integer value designating success or failure **Error:**
ParseArg	**Description:** Takes a char* argument and verifies that it follows that name/value syntax defined as <name>=<value> **Input:** Next char* argument on the list **Output:** **NameValuePair**. NULL will be returned in the event of a syntax error **Error:**
ProcessArg	**Description:** This routine performs the semantic analysis of an argument. 1. Look up name in the **ArgTbl** 2. Verify that the value is valid 3. Add the name value pair to a list of processed arguments called **ArgVec** list. 4. If this name value pair already exists in the list then issue a diagnostic. 5. Call the supplied processing function for this argument as specified in the **ArgTbl** **Input:** **NameValuePair** **Output:** Interger value designating success or failure (0 for success, positive integer for other errors) **Error:**
ParseName	**Description:** Verify that the Name part of the argument conforms to being alpha-numeric **Input:** char* Name part of argument **Output:** char* Name else NULL **Error:** None
ParseValue	**Description:** Verify that the Value part of the argument conforms to being alpha-numeric and/or punctuation characters **Input:** char* Value part of argument **Output:** char* Value else NULL **Error:** None
FinalizeArgs	**Description:** Post process the argument list. The purpose of this routine is to validate that all required arguments have been defined on the command line. Also processes and adds default arguments to the **ArgVec**. **Input:** None **Output:** Success or Failure **Error:**

Configuration Manager:

```
class  Tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};

class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > ConfigArray;

class ServerConfig {
private:
        ConfigArray Array;
public:
        ServerConfig(serverId, dsn, dbuser, dbpasswd); // Initialize from db
        ServerConfig(serverId, string filename); // To initialize from a file.

        ConfigArray* GetConfigArray(int serverId);
        Tuple* FindConfig(string Name);
        int Reload(void);
        Tuple* GetConfig(int serverId ,string Name);
};
```

ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes configuration manager. 2. Opens the database and gets configuration array **Input:** Server Id, Data Source Name, Database User name, and database users password. **Output:** None **Errors:**
ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes Configuration Manager. 2. Opens configuration file and reads configuration array. **Input:** filename of flat-file configuration. **Output:** None **Errors:**

GetConfigArray	**Description:** Returns the entire configuration for a given server id. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying which server to retrieve configuration for **Output:** Returns a vector holding the configuration or NULL **Errors:**

GetConfig	**Description:** Returns the configuration for the specified name. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying the server to retrieve configuration for and Name of configuration item. **Output:** Configuration Tuple. A Tuple may be a nested Tuple. NULL if an error is encountered. **Errors:**

FindConfig	**Description:** Returns the Tuple specified by the name. This routine does not go to the database or flat-file to get its value. Rather it finds the value in the ConfigArray maintained by the Configuration Manager. **Input:** Name of the configuration item. **Output:** Configuration Tuple. NULL if an error is encountered or the Tuple does not exist in the current configuration. **Errors:**

Reload	**Description:** Reloads the entire configuration from the database or flat-file. This routine may reload its configuration indirectly through the use of the System Monitor. In this case it will make a message request to the monitor and listen for the configuration results. **Input:** None **Output:** integer specifying success or failure. Zero will be returned in the case of Success. A negative value in case of error. **Errors:**

Logging Utilities:

```
class LogManager
{
private:
        char* FileName;
        int MaxFileSize;
        char* ResourceFile; // message catalog file

        char* GetMessage(MsgNum, MsgStr)
public:
        LogManager(ServerId,Size=10);
        LogMessage(MsgStr);
        LogMessage(ThreadId, MsgNum, MsgStr, ...);
```

```
};
```

LogMessage	**Description:** Write message out to log file. There are two forms of LogMessage. The first will write out a message buffer as is (unformatted) bypassing the resource file. The second form will format the message. Both forms of LogMessage always pre-append a time stamp. 1. Lookup message number in the resource file and get message string 2. format the log message using time stamp, thread id, etc. 3. write out message into the log file. **Input:** Thread Id, Message Number, Message String, and variable number of arguments. **Output:** None. **Error:**

GetMessage	**Description:** Routine returns a message string from the resource file for the message number specified. **Input:** Message number, C Locale text string. **Output:** Message string. Either way, Get Message will always pass a return a valid message string by either returning the string from the resource file or by passing back the MsgStr passed in. **Error:** If an error occurs trying to get a message from the resource file, a message will be logged to the error log.

```
class ErrorLog: protected LogManager
{
private:
        LogLevel ErrorLogLevel;
public:
        ErrorLog(ServerId, LogLevel=0, Size=10);
        LogError(ThreadId, ErrorNum, ErrorMsgStr, ...);
};
```

LogError	**Description:** Writes output to error log file. 1. Check that the message level against the current ErrorLogLevel. 2. Format the message and call the long form of **LogMessage** to write the buffer out to the file. **Input:** 1. ThreadId: Thread identifier to help with the debugging process. 2. ErrorNum: Error number used to uniquely identify an error message in the resource file. 3. ErrorMsgStr: Message string which includes stdio like string formatting. 4. …: variable list of arguments to be inserted into the message string per the format. **Output:** None. **Error:**

Testing design

Each of the components that make up the Server Component Framework will be able to be tested independently of the other components. Each component will have a main entry point defined within a testing .exe to accomplish the Unit testing phase.

Testing of the component framework will be done in phases. Each of the phases is described below along with its dependencies.

Phase 1: Unit testing Test basic components that make up the framework. Each components functionality, restrictions, and boundary conditions will be tested. Will allow testing common configurations for a single server component. This round of unit testing will test the integrated component utilities and framework.	**Dependencies:** 1. ServerComponent class 2. ServerStateMgr class 3. ArgList class 4. Logging Utilities 5. Configuration Manager (flat-file)
Phase 2: Unit testing (full functionality) Test full functionality including messaging interfaces and database connectivity.	**Dependencies:** 1. Phase 1 2. Database connectivity 3. Messaging Service
Phase 3: Integration Testing	**Dependencies:** 1. Phase 2 2. System Monitor (including backup) 3. SLiM Server, App Server, Web-Server
Phase 4: Stress Testing See section on stress testing for details	**Dependencies:** 1. Phase 3

Unit testing plans

Command Line Utilities

The Command line utilities will be tested in a stand-alone module called cmdline.exe. It will support the command line arguments defined in this document.

Configuration Manager

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest.exe executable should be testable in the DB and the non-DB mode.

Logging Utilities

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Server State Manager

The Server State Manager and the Server Component Framework will be tested independently of specialized components. The routines that require specialization (**StartProcessing**, **StopProcessing**, **HandleError** and **UpdateConfig**) will be provided to simply return successfully.

Stress testing plans

Stress testing will require having at least the System Monitor functionality implemented since it is used to drive the server components.

1. Test to repeatedly start, stop, reconfigure the server component.
2. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
3. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
4. Test lost database connectivity
5. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
6. Test error recovery under adverse conditions.
7. Test recovery from running out of memory, thread resources.
8. Test recovery from threads dying.
9. etc.

Coverage testing plans

1. Goal: 100% path flow coverage. Only exceptions for known error conditions that cannot be practically reached (e.g. thread synchronization, etc.)

Cross-component testing plans

The following pair-wise testing will be performed:

1. framework/database (phase 2)
2. framework/messaging (phase 2)
3. framework (System Monitor) /framework (backup Monitor) (phase 3)
4. framework/Web Server (phase 3)
5. framework (System Monitor) /framework (Other Servers) (phase 3)

Upgrading/Supportability/Deployment design

1. Each error condition will be documented with explanations and practical work-arounds
2. Component framework will support enhanced debug option to dump additional de-bugging information to special log files.

Open Issues

eStream System Monitor Low Level Design

Michael Beckmann
[redacted]

Functionality

The role of the System Monitor is to monitor the state of the Application Servers and SLiM servers within an eStream deployment. In addition, it also manages a back-up System Monitor.

1. The System Monitor provides the following key services:
   a. Monitors and reports server load across machines
   b. Monitors server state across machines.
   c. Acts as a communication conduit to the database for configuration information needed by the server components.
   d. Initiates state changes within the server.
      i. Start/Stop servers
      ii. Sends requests for servers to update their configurations
2. The system monitor runs as it's own process. Within a multi-system deployment, there will be at least two monitoring processes, each on a different machine:
   a. One monitoring process will act as a primary and the others as backups.
   b. In the event that the primary monitor goes down, one of the backups will take over the primary monitoring responsibilities.
3. The monitoring process can re-launch a server side process if a process terminates unexpectedly (this is a configurable option).
4. The system monitor manages server state by maintaining regular communication via a heart beat protocol between itself, the backup monitors, and every logical server.
5. The system monitor's heart beat supports a light-weight messaging protocol between components to initiate state changes.
   a. Simple request pulse.
   b. Stop request pulse
   c. Configuration request pulse
6. The monitor will raise an alarm if it does not receive a heart beat response from the servers within a specified period of time.
7. The system monitor's heart beat request rate is a dynamically configurable parameter maintained within the database.
   a. The rate can be changed through the administrative UI.

System Monitor Component Overview:

The System Monitor is made up of the following distinct components:

1. Server Component Framework (described in detail in the Server Component Framework Low Level Design)
    a. Server State Manager
    b. Configuration Manager
    c. Messaging Utility
    d. Logging Utility
    e. Command line Utilities
    f. Thread Management Utilities
2. Load Monitor
3. Server Component Manager
    a. Remote Process Launching Utilities
    b. Heart-beat protocol
4. Database Request Manager
    a. Database request protocol

Server Component Framework:

The System Monitor is extended from *the Server Component Framework* by providing an implementation for the following plug-able interfaces: **StartProcessing, StopProcessing, UpdateConfig,** and **HandleError.** Each of the interfaces is described in detail below. The routines **StartServer** and **StopServer**, which are also part of the interface are inherited from the Server Component Framework and do not need to be provided by the System Monitor.

System Monitor State Transitions:

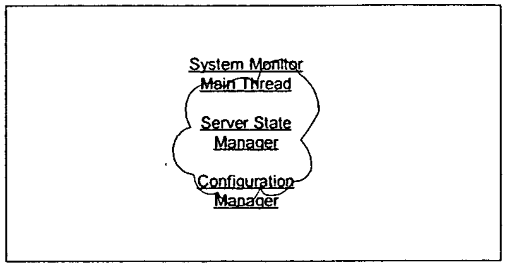
Since the System Monitor is extended from the Server Component Framework, it needs to provide the four interface implementations called out above for each state transition. This section gives a brief overview of the primary state transitions and what is going on within the System Monitor for each transition.

STOPPED -> IDLE:

When the System Monitor is initially started by the WebServer it transitions from the **STOPPED** state to the **IDLE** state via a call to the **StartServer** routine. In this transition the following events occur:

1. Configuration manager reads common configuration from the database.
2. Apply configuration and initialize (System Monitor)
3. Change state to **IDLE** state.
4. Send acknowledgment

In the **IDLE** state the System Monitor is still maintaining only its main thread which it inherits from the Server Component Framework and runs the Server State Manager, Configuration Manager, and the Messaging Service:

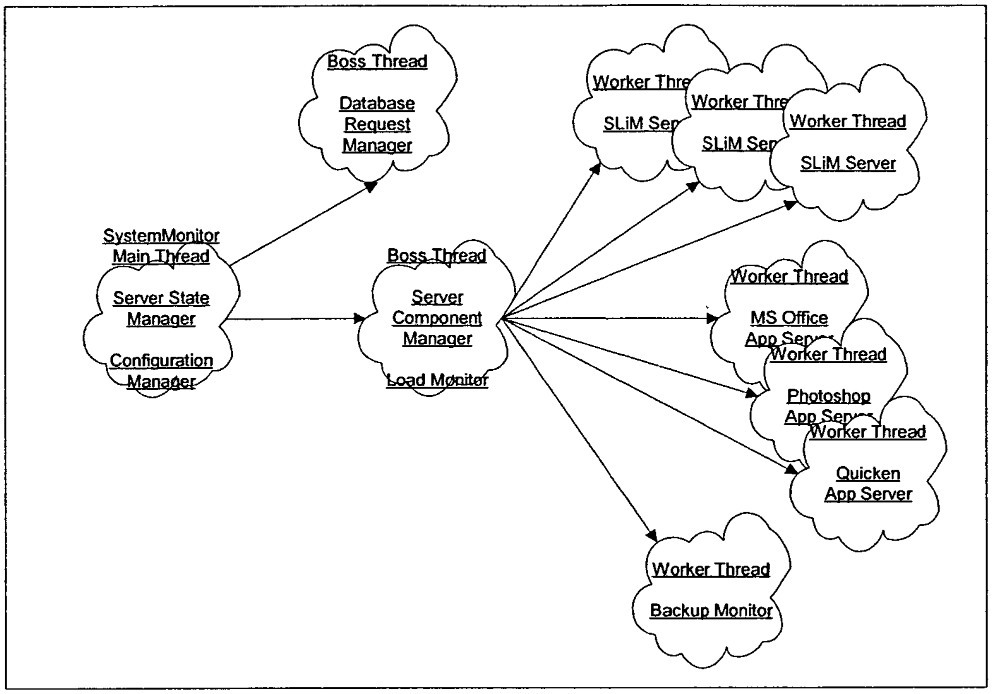


IDLE->PROCESSING

When the System Monitor transitions from the **IDLE** state into the **PROCESSING** state via a call to **StartProcessing** it creates a number of processing threads to do its work. In **PROCESSING** mode the System Monitor employs a "Boss-Worker" parallel programming model:

1. The System Monitor creates two processing threads that acts as the "boss" threads.
2. The first thread is for the *Server Component Manager* to manage each and every server in the deployment. The thread entry point is **MonitorServers**
3. The second boss thread is used for the *Database Request Manager* to service database requests from each of the server components for configuration and any special application server requests. The thread entry point is **ProcessDbRequests**
4. The boss thread *(Server Component Manager)* reads the common configuration for all server components and spawns worker threads for each logical server component. A thread will be allocated to monitor each logical server processes. The thread entry point for each of the worker threads is **ManageProcess**
5. The *Load Monitor* is initialized and runs as a service within the Server Component Manager boss thread.

Below is an illustration of the thread model/architecture when the System Monitor is **PROCESSING** state.



Load Monitor:

The *Load Monitor's* role is to aggregate the load information for each server component within the deployment and update the information to the database.

The *Load Monitor* receives regular load information from each of the server component processes via the *Server Component Manager*. The load information is provided in the acknowledgement to regular heart-beat requests that each of the worker threads initiates with the server process that it is managing.

The *Load Monitor* runs in the "boss" processing thread of the Server Component Manager. The Load Monitor provides thread safe interfaces.

For each application being served within the deployment the *Load Monitor* maintains a list of servers and their response times. It periodically updates the database with these lists which are then consumed by the SLiM servers. The frequency at which it updates the database is configurable.

Server Component Manager:

The System Monitor manages each and every server component process within a deployment. This management activity is undertaken by the *Server Component Manager*.

The *Server Component Manager* spawns a worker thread for each server components within a deployment. The primary entry point for each of the worker threads is **ManageProcess**. This routine has the responsibility of starting and stopping each process and tracking its state, as well as requesting state changes in the component it is managing.

Launching and Terminating Remote Processes:

The System Monitor has the responsibility of launching server component processes across the deployment. The deployment can be made up of a heterogeneous set of servers. In order to accomplish this the system monitor communicates with either a Unix daemon or an NT service that runs on every server within the installation.

The service/daemon is a very simple process that listens on its connection for requests to launch or terminate a process.

If the daemon dies or is accidentally killed, the system will automatically re-launch the daemon. There are facilities for this on Unix and on NT.

The System Monitor communicates with the service via the EMS utilities and a unique protocol defined below in the interfaces section.

Once a process is launched the daemon/service does not track status or trap output of the process (fire and forget). Tracking of the process will be initiated by the System Monitor via the heart-beat protocol.

Database Request Manager:

In order to save database licensing costs, it has been decided that the Application Servers not have direct database connectivity. Instead, the Application Servers will depend on the System Monitor as an intermediary to read and write data from the system database.

The System Monitor is used on behalf of all the other servers to interface with the database for retrieving configuration information and/or any other information that the Application servers may need.

The Database Request Manager is launched from the **StartProcessing** interface. It is launched in its own boss thread. The thread entry point is **ProcessDbRequests**.

The Database Request Manager also defines a protocol for each of the database interfaces defined in the WebServer/Database interface document.

Data type definitions

The System Monitor maintains a few key data structures beyond what is provided by the *Server Component Framework*:

Request Queue:

The System Monitor makes use of mutex protected queues to communicate between threads. For example, each worker thread will maintain its own mutex protected input queue. The following components will maintain input queues:

1. Boss thread for Server Component Manager
2. Boss thread for Database Request Manager
3. Each server monitor worker thread

A queue entry models, fairly closely, the data contained within the messaging protocol between servers as defined in the *Server Component Framework*. This data can be used for the input queues for each worker thread managing individual work components.

The details of the input queues themselves is defined in the common thread API.

```
struct QueueEntry {
        int Request
        ServerID SenderID
        ServerID RecieverID
        ServerID TargetID
        Data
}
```

Managing Worker Threads:

The System Monitor's boss thread **(MonitorServers)** maintains an in memory list of all known servers within a deployment, and their associated worker thread ID. In addition, it maintains information about request queues for each worker thread. This list is maintained as the **ServerInfoList**. The list is made up of entries which maintains all the necessary information to start/stop/manage/communicate/etc. with a server component. The **ServerInfoList** is traversed by the Server Component Manager and the Load Monitor.

```
struct ServerInfoEntry
{
        ServerConfig* Config;        //configuration  for each server component
        Thread ThreadID;             //Thread ID in which the Server Component
                                     //Manager is executing
        ThreadQueue* InputQueue; // processing queue for this thread (defined in thread
                                     // utilities package.
};

typedef vector <ServerInfoEntry> ServerInfoList;
```

Load Monitor:

The Load Monitor maintains the following data structures:

```
struct ServerLoadEntry
{
        ServerID
        LoadInfo  // still needs to be defined
};
struct ServerSetEntry
{
        int AppID
        list <ServerLoadEntry> ServerSet // ordered list of application servers
};
typedef vector <ServerSetEntry> LoadMonitorList;

class LoadMonitor            // thread safe/ re-entrant
{
private:
        LoadMonitorList List;
public:
        LoadMonitor();
        ~LoadMonitor();
        UpdateLoad(ServerID, LoadInfo);
};
```

Server Monitor:

```
int MonitorServers(void);  // non-member function used as thread entry point
int ManageProcess(vector <ServerInfoEntry>); // non-member function used as entry
                                              // to for worker threads
```

System Monitor Configurations:

Configurations unique to System Monitors include:

Information	Supports Dynamic Config	State	Description
LoadUpdateRate	Yes	All	Frequency at which the Load Monitor will update the database with aggregate load information. Value is specified in milliseconds. If the value is zero then the Load Monitor will not update the database essentially disabling load monitoring.

Database Interface Protocol:
There will be a unique operation code for each database interface. Likewise there will be an acknowledgement.

OpCode	senderID	receiverID	targetID	Data

Interface definitions

Server Component Framework:

The implementation of the System Monitor is derived from the generic *ServerComponent* class. Refer to the *Server Component Framework* Low Level Design.

Implementing a *SeverComponent* requires the definition of the following methods:

StartServer	**Description:** This routine is called to start a server process and bring it to the **IDLE** state. It assumes that the server is in the **STOPPED** state. This routine performs all the necessary initialization and configuration common to all server components. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StopServer	**Description:** The routine StopServer is called to cleanup and terminate a server process. It assumes that the server component is in its **IDLE** state and is therefore not processing any requests. This function is predefined in the *ServerComponent* base class. The System Monitor has no need to override it. **Input:** None. **Output:** Integer value designating return status. Success = 0. **Errors:**

StartProcessing	**Description:** This routine initiates all the activities unique to the System Monitor: 1. Launches a boss thread. Call **MonitorServers** as the boss thread entry point. 2. Launch another boss thread. Call **ProcessDbRequests** as the primary entry point. 3. Return immediately Note: The **StartProcessing** routine must return immediately else the Server State Manager will be blocked. **Input:** None. **Output:** Integer value designating return status. Zero if success else error **Errors:** Error launching thread.

StopProcessing	**Description:** This routine will perform the following activities: 1. Terminates each server component's monitoring thread. This effectively disables the monitor from managing any executing server components including stopping existing components or starting new ones or servicing configuration requests. 2. Terminates the Database Request Manager. Note: The primary system monitor will only execute this interface if a system administrator explicitly changes the monitors state or an error has occurred within the monitor. **Input:** None. **Output:** Integer value designating return status. **Errors:**

UpdateConfig	**Description:** This routine will perform configuration changes specific to system monitor functionality. Refer to the section on system monitor configurations. This routine can be executed from both the IDLE and PROCESSING states. **Input:** None. **Output:** Integer value designating success or failure. Zero for success. **Errors:**

HandleError	**Description:** This routine is called by the Server State Manager whenever an error is encountered. If this routine can handle the error then the original calling state will be maintained. If the error cannot be handled the process will transition to an ERROR state and terminate. **Input:** None **Output:** Integer value designating success or failure. zero for success. **Errors:** Error specifying that the error could not be handled.

Server Monitor:

MonitorServers	**Description:** This routine is defined as a non-member function and is the thread entry point for the "boss" thread of the Server Component Manager. It is the primary thread for managing and monitoring all the server components. 1. Create an input request queue for this thread 2. Start Load Monitor 3. Go to the database and get a list of all the server components calling **GetServers** as defined in the Database Low Level design document. 4. For each server component retrieve its common configurations calling **GetServerConfig** and create and entry into the **ServerInfoList** 5. Spawn a worker thread to manage/monitor the backup System Monitor. Call **ManageProcess** as the thread entry point passing in the **ServerInfoEntry** After the backup monitor has been launched, repeat step 4 for each server component. 6. Check the input queue for new worker requests. 7. Loop back and redo steps 3-6 looking for new servers to manage **Input:** None. **Output:** **Errors:**

ManageProcess	**Description:** Non-member function that launches and monitors the specified server component processes. It is expected to be the start routine for its own thread. This routine is called from **MonitorServers.** Performs the following steps: 1. Call **StartRemoteProcess** to launch the server component per the configuration information provided in the **ServerInfoEntry**. 2. Open a point-to-point messaging connection to the newly launched server component. 3. Initiate a regular heart-beat request at the rate defined within the configuration. 4. Enter monitoring loop a. Check for requests from the input queue for this server component. b. Send the request to the server currently being monitored by this thread. c. Listen and wait for response; timeout if wait is too long. d. If a response is received update state information in **ServerInfoEntry** and update the database as necessary calling **SetServerState** and **SetServerLog.** e. Enqueue load information calling **UpdateLoad** f. repeat a-f **Input:** **ServerInfoEntry** **Output:** Status of the server component on exit. **Errors:** 1. launch error 2. messaging error 3. unexpected server component termination

Database Request Manager:

ProcessDbRequests	**Description:** This routine is the thread entry point for the Database Request Manager. It is called from **StartProcessing**. 1. Connect to Database 2. Create a listener using EMS utility 3. Sleep until a request comes in and then enqueue the request. 4. Worker threads will then dequeue and satisfy the db requests and terminate **Input:** db connectivity info **Output:** **Errors:**

Load Monitor:

The *Load Monitor* maintains three public interfaces:

UpdateLoad	**Description:** This routine updates, in memory, server sets with the current load information. This routine is thread safe/re-entrant This routine is called by **ManageProcess** 1. Traverse server lists by application. If the application. If the application entry is not found create it and add it to the list. 2. If the application entry is found search for the server entry 3. Update load information 4. Call **FlushLoadData** per the database access frequency configuration. **Input:** Load information, ServerId **Output:** Integer value specifying Success or Failure. Zero if success. **Errors:** Error processing load information
LoadMonitor	**Description:** Initialization routine for the Load Monitor. Called by **MonitorServers** 1. Initialize server lists to NULL. **Input:** None. **Output:** None. **Errors:**
~LoadMonitor	**Description:** Destructor for the LoadMonitor. 1. For each server flush all load data to the database calling **FlushLoadData** 2. Cleanup server set data structures and exit. **Input:** None. **Output:** None **Errors:**

Primary and Backup System Monitor:

The backup System Monitor needs to be able to take over the primary system monitor responsibilities in the event that the backup loses communication with the primary. The following interface will cause the backup monitor to take over as primary.

SwitchPrimaryMonitor	**Description:** Take over primary monitor responsibilities. This routine is called from the **HandleError** routine defined for a backup monitor. If the backup monitor doesn't hear from the primary monitor, it assumes that the primary monitor has died. 1. Update database validating monitor switch. 2. If no database connectivity then go back to listening mode until the next time out. 3. If database connectivity then attempt to shut down the old primary System Monitor just in case it is still running. 4. Go through the same steps as the primary monitor would do when its **StartProcessing** routine is called including starting up a new backup system monitor **Input:** None **Output:** integer value designating success or failure **Error:**

Launching and Terminating Server Processes:

TerminateRemoteProcess	**Description:** This routine terminates a remote process either on a local machine or on a remote machine. This routine is called by **ManageProcess**. 1. Verifies that the target machine is up and running 2. Send message to daemon/service to terminate the process. Uses EMS messaging service. **Input:** Process info, target machine **Output:** integer return specifying success or failure. **Errors:**

StartRemoteProcess	**Description:** This routine spawns the requested process either on the local machine or on a remote machine. This routine is called by **ManageProcess** 1. Verifies that the target machine is up and running by sending ping to daemon/service on the specified machine. 2. Send message to daemon/service to launch a process passing the process path and any arguments. **Input:** Process name, target machine, attributes **Output:** integer return where 0 designates success else error code is returned. If successful the routine will also return process info **Errors:** 1. machine not responding 2. executable not found 3. failure on launch

Daemon/Service	**Description:** Unix daemon/NT service which runs on every server machine within the deployment. Its role is to dispatch requests to create and terminate processes, obtain process information. 1. Initialize/Create a listener using EMS 2. Go to sleep until a request comes in 3. Dispatch request 4. Return information to requestor 5. Perform any necessary logging 6. Repeat 2-6 **Input:** listening port **Output:** None **Errors:** Errors will be written to a special log file

Testing design

The System Monitor will provide a command line option to facilitate testing the component in its various testing phases. The option will allow the system to manage its interfaces through flat files.

Unit testing plans

To unit test the System Monitor it will require a single generic server component running on the same machine.

Phase	Description	Dependencies
Phase I: Local process management	Cycle through all the state changes for a single local generic server component	1. Functionally complete Common Server Framework 2. Generic Server component
Phase II: Remote process management	Same as Phase I for remote processes	1. Phase 1 dependencies 2. Server daemons
Phase III: backup monitor failover	Verify that the backup monitor fails over and continues to manage the system	1. Phase II dependencies 2. Backup Monitor Server

Stress testing plans

Tests should include:

1. Max # of generic server components on a single machine.
2. Max # of generic components across multiple machines.
3. Max # of generic servers changing state at least once per second
4. Test to repeatedly start, stop, reconfigure each server component.
5. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
6. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. Auto Restart.
7. Test lost database connectivity
8. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
9. Test error recovery under adverse conditions.
10. Test recovery from running out of memory, thread resources.

11. Test recovery from threads dying.

Coverage testing plans

The System Monitor will achieve 100% code coverage with the exception of error conditions which are possible however difficult to reach in practice.

Cross-component testing plans

The System Monitor interacts with the following components:

1. System Monitor/Database
2. System Monitor/WebServer
3. System Monitor/Server Component Framework (other servers)
4. System Monitor/Backup Monitor

Upgrading/Supportability/Deployment design

1. All diagnostics will be documented as to their root cause and workarounds/actions to be taken.
2. The system monitor will support an enhanced debug support which dumps additional information to special debug logs.

Open Issues

1. Need to identify a mechanism to ensure that we do not have more than one primary monitor running at any given time.

eStream Messaging Service Low Level Design

Version 1.1
Sameer Panwar
[redacted]

Functionality

Estream is fundamentally a networked system, and thus must rely on a communication infrastructure as part of its foundation. The EMS subsystem provides the API for eStream processes to communicate with remote eStream processes. It is subdivided into layers and is intended to freely support different combinations of choices from each layer in an extensible manner. E.g. XDR over HTTP over SSL, while new layers can be added in the future (for compression, custom encryption, etc.). This is important considering it will be used across the variety of eStream components, including the client cache manager and all the servers.

The layers are enumerated as follows:

1. Packaging Utility [XDR/SOAP/name-value pairs, etc.]
2. Procedure Call Layer (PCL)
3. User Layer(s)
4. Message Transport Layer (MTL) [SSL+TCP/HTTP+TCP/MSG+TCP]

The Packaging Utility is primarily used to pack and unpack the parameters of the message into the body. It isn't really a true layer, in that it is visible to the other layers—they also use it to pack their data into their headers (except if the layer is a given standard). Thus all layers are subject to the same packaging scheme, unless they provide their own internally. The various packaging utilities won't be tied to EMS directly—they will be available for packaging data into any buffer.

The Procedure Call layer provides the functionality to assign API and Function identifiers in order to define a remote API, as well as providing a dispatch mechanism on the server side to enable appropriate processing of incoming messages. This layer will also handle callbacks for replies to asynchronous messages. The implementation for eStream 1.0 for this layer will be called EPC for eStream Procedure Call.

The User layer is present to allow the user of EMS to add custom headers on the send side and perform early checks (e.g. on AccessTokens) on the receive side before being queued for service in the thread pool. This could also be used for compression or a custom security layer, and multiple User layers could be put in place.

The Message Transport Layer is special in that it is this layer that uses the actual socket interface. However, it has the additional functionality (referred to as MSG above) to recognize message boundaries via a magic number and a message size (which is also done

via an HTTP header or an SSL header). This enables us to grab a complete message over the stream-oriented TCP before sending it up to higher layers. Note that UDP might not be supportable in EMS—there may be inherent assumptions that the underlying network protocol is reliable and connection oriented, but I won't know for sure until EMS is implemented for TCP whether it will be adaptable to UDP. Also, the MTL is also responsible for contacting proxy servers in the manner appropriate to the protocol they're using (SSL or HTTP).

Messages are linked lists of memory blocks (which also track size used), each block being a layer header, with the original message body last. They are always flattened in MTL before sending, but can be flattened elsewhere if desired. Probably we'll use 4k blocks for each header, and a 64k block for the body. This linked list of blocks is called the Message Buffer, which is contained in the Message object. Messages also contain per-message info relevant to the various layers, including API & FUNC #'s, Message IDs, etc. This per-message info is used to generate the appropriate headers for outgoing messages, and is unpacked into the Message for incoming messages. It is the sum of all per-message info any layer might want, placed in the single Message class, even though obviously only the layers in use for the given connection will actually be active.

EMS layers have:

- Header data members
- BuildHeader function
- BodyXform transformation function (e.g. compression, but often just null)
- Send function
- Internal Connection info (nonces, request ids, SSL_CTX, etc.)
- Reset function (when need to re-establish a connection)
- Session info that's global for the layer, static data (session ids, callbacks, etc.)

In usage, each layer, in the outgoing direction

1. Gets outgoing Message, becomes owner of the memory.
2. Builds a header (needs to grab a free block) using per-message info set up by the user in the Message, and adds it to the Message Buffer.
3. Applies any applicable transformation to the message body.
4. Passes the Message it to the lower layer.
5. Each layer is NOT to remember anything on a per message basis, since it won't be notified if the message is lost!

Each layer, in the incoming direction

1. Gets the incoming Message.
2. Unpacks the header, performs any relevant checks and transformations on the message body (that is, the body relative to this layer's header). Increments a pointer so that a higher layer can know where its header starts.
3. Places any relevant header info into the per-message info in the Message, e.g. Message IDs. This layer-related info will thus be available outside EMS. Only a limited number of these will actually get set, since some of these are only relevant in the outgoing direction.

4. Passes the Message (or an error) to the higher layer. If the higher layer is the EMS caller, then the Message is delivered to them so that they can then unpack their data.

Steps to use EMS:

1. First, a Connection object must be initialized, describing the remote server, the stack of layers and packaging utility, along with other configuration.
2. Next, the user must package their data into a Message Buffer.
3. Then they must set up any required per-message info in the Message object.
4. Finally, they can send the Message (this triggers the actual layer processing, and then the Message gets queued to go out the network).

Message Stack

A Message Stack is an object that describes the order of the actual layers being used for a given remote server, and also binds to a packaging utility, which is used by the layers and user to marshal and unmarshal data. The Message Stack doesn't actually contain the layer objects themselves, just pointers, so there is just one Message Stack object. For any listening port on a server, only one Message Stack is used, and essentially defines the language spoken on that port. Other ports on the same server can use different Message Stacks.

Connection

The Connection object contains all relevant data EMS requires to manage a connection with a remove server. It contains the relevant layer objects, which keep track of connection-level info, as well as a Message Stack. It also contains the remote server's DNS name and/or IP address and port number, and the same for a proxy server if necessary, these being used by the Message Transfer Layer. The reply timeout and socket handle and state are also maintained here. There is one Connection object for every combination of layers (and packaging utility) and remote server we care to use, and instantiates each layer object and initializes it (and assembles the Message Stack) as part of its constructor. A given connection will be configured to support synchronous or asynchronous communications, but not both. A Connection also contains an outgoing message queue, and a (pointer to a) buffer for messages in the process of being read or written to the network along with a current position index.

Connection Management

I/O can be nonblocking, but the first 3 layers **are not allowed to block on an OS call**, so stuff only gets queued in the MTL. Thus there is one well defined place where sends/receives of messages can be resumed, and the states involved are described below. If the socket is blocking, then we need to set a timeout and return an error on a failure.

A special thread called the socket watcher does a select on active sockets, and triggers actions based on the socket's connection state and message state. If the state isn't CONNECTED, then it continues the relevant handshake. If it is CONNECTED, then it sends

the outgoing message, or dispatches an incoming one. It is also responsible for handling timeouts.

Connection states:

DISCONNECTED	
TCP_HANDSHAKE_WAIT	select(write/except)
MTL_HANDSHAKE_R_WAIT	select(read)
MTL_HANDSHAKE_W_WAIT	select(write)
CONNECTED	select(read)

Message state (outgoing) for each connection:

EMPTY	select(read)
MSG_READY	select(read/write)
MSG_SENDING	select(read/write)

Relationships:
1 Connection Object per remote server, and 1 associated Message Stack for the Connection Object. Thus, if communication to the same server is required using two different Message Stacks, then 2 Connection Objects are required. Multiple Connection Objects with the same Message Stack to the same server are allowed for clients, but discouraged, since that eats up server resources.

<u>Handshakes</u>
Some of the underlying protocols need to perform handshakes before they can exchange user data. E.g. SSL needs to perform authentication and key exchange and obtain a session before the user's data can be encrypted for transmission. These handshakes can take more than one exchange to establish the session, e.g. SSL takes 2 ½ round trips (on top of TCP's 1 ½ round trips). SSL's interface for handshaking supports nonblocking I/O by implementing an internal state machine to handle the handshake protocol. EMS will export this interface on the send side (by keeping a connection state); for receives, EMS will have to check to see what the connection is waiting on (read or write, even if the user request was different) and check for it during a select(). If data is returned when the connection is still being set up, then the handshake is continued to read the waiting data. After the handshake is complete, the layer must check for any pending message, process it with the session data, and pass it down. While a handshake is going on, all new requests are processed down to the MTL and then queued. This means that header data members relating to the MTL cannot be changed, so access to them will be through functions which will fail on a call for a given connection object that is not disconnected. Handshakes are NOT supported at any higher layer than the MTL, since that would entail too much complexity. Thus any other desired handshaking must be implemented on top of EMS, though handshaking could be possibly supported within higher layers if communication were only synchronous, if the need arises. A successful SSL handshake will set the session ID. If the handshake fails on trying to re-use an old session ID that expired, then the session ID is cleared, and a full handshake is required again.

<u>Global EMS Stuff</u>

EMS at the global level manages the messages being sent. When timeouts are being used, EMS can assume that for every outgoing message some response message must be returned from the server, and will report a timeout to the caller if a response is not received by the deadline. EMS requires a Message ID (also used by the PCL dispatcher) to be passed in by the user, which it stamps on the outgoing message, and is present on the reply message which cancels the timeout. Message IDs are also used by the PCL dispatcher, so that the client callback can match the response with its request. EMS requires that the user use a monotonically increasing Message ID, and will thus reject any reuse (this enforces uniqueness of Message IDs for the lifetime of the process). Since Message IDs are low-level EMS details, they live in the MTL. EMS keeps an ordered queue of Message IDs and timeouts (but throws away the actual message contents once it's sent), and the socket watcher thread uses the earliest timeout in its select() call.

<u>Asynchronous calls</u>
For asynchronous calls, client callbacks are registered one for each remote function (these are managed by the PCL), as well as a timeout function (which is handled by the MTL). The socket watcher thread gets messages as they come in, and the PCL will queue them for a dispatcher thread to process.

<u>Buffer/Memory Management</u>
Since we need to manage buffers efficiently, and also avoid possible attacks if we support very large message sizes, we should pick some reasonable maximum size for any message. For now, I will set that to be 64k (remember, this is the max size **after** packaging), which really may only affect the app server. This maximum size, of course, includes headers in any of our layers, and may be restricted even more when SSL is in use (due to its own restrictions). We should also consider the lifetime of memory buffers. In time sequence, normally what happens is that the user grabs a buffer for the main message body, packs their data into it, then passes it to the top EMS layer. Each EMS layer then grabs a buffer for the header, and at the last step this is flattened into yet another buffer (which frees the previous buffers), which is then passed to the network, after which the Message object can be freed. When a message is received, it is placed into a buffer in a Message object, which is given to the user, who is then responsible for freeing the Message object when they're done with it. Before that, the unpackaging functions may allocate memory for variable sized objects, which then also become the user's responsibility to free.

Since the lifetime of many of the memory buffers is short, but the number of outstanding messages is generally pretty low, reuse of buffers can definitely help efficiency. For the various transient classes, I expect to implement a memory manager class that will keep the objects around instead of freeing the memory, and then reuse them when a new one is needed. Thus each object will have its own pool. Buffers will be managed the same way, probably coming in 4k (for headers) and 64k sizes. However, I'm not sure if I want to use this approach for memory allocated for the variable sized objects in the unpackaging functions; maybe I'll just use malloc for that.

<u>Socket Watcher</u>

The socket watcher thread is required to support asynchronous messaging, by grabbing and dispatching messages as they come in, also making it the logical place to handle timeouts for asynchronous threads. This special thread needs two data structures—the list of sockets to watch and a queue of timeouts. Connections that are configured as synchronous never pass their sockets to the socket watcher. For asynchronous connections, whenever the connection state changes, the socket may be added/updated in the socket watcher's list (we need to know whether to check if the socket can be read from or maybe written to, depending on the connection state). When the MTL discovers that the TCP connection was closed, it removes it from the socket watcher's list. Note that the socket watcher is actually "just below" the MTL: it doesn't know how to decode a Message ID. When a socket is ready for reading, it gets the associated Connection object, and then calls the MTL (in the socket watcher's context) to pull the message out and then cancel the pending timeout. Then, still in the socket watcher thread, the message is pushed up the Message Stack until it is dispatched to another thread for actual servicing (as in the PCL). Then the socket watcher goes on to process any other ready sockets. Note that thus the socket watcher could process messages going to different MTL's. There is only a single socket watcher thread in EMS for a given process. The socket watcher also checks for incoming connections on the listening ports (there can be more than one on a server).

The socket watcher is also responsible for completing sends of messages that were left incomplete (i.e. would have blocked) or are queued. A connection's message state reflects this and has the watcher check for socket writability—when that happens, it then tries to complete the send as appropriate. In fact, if asynchronous messaging is being used, all actual writes over the network happen in the socket watcher thread, the user threads just dumping the messages into the queue and changing the connection state appropriately.

When a user uses synchronous messaging, this is different in that the user's own thread then sends the outgoing message, waiting until the send is complete, and then does a select on just that one socket with its own timeout. When the reply is received, then the MTL and higher layers are executed in the user's context, and no dispatch to another thread occurs (since the Connection is configured as synchronous).

<u>Clients vs. Servers</u>

Client and Server behavior is distinguished by the EMS calls they use. Clients use EMSMsgSend(...) which takes a Message ID and a timeout value, and if they're making asynch calls, then they must also set up callbacks. Servers use EMSMsgReply(...) which returns the above Message ID and has no timeout. Despite this difference, the socket watcher behaves the same on the client and the server. It simply does a select on all open connections and tracks timeouts. Of course, the server just disables the timeouts. Also, the server listens on a specific socket for incoming connections, while the client does not. However, servers have a different kind of timeout—the connection idle timeout, which means if a connection isn't used by a client for a certain interval, the server will drop the connection to conserve resources (sockets). The server's MTL will establish its own

callback with the socket watcher in order to close the connection when it times out (and it will also cancel timeouts whenever it receives a complete Message). The client's MTL will transparently detect the dropped connection and reestablish it when the next outgoing message is queued.

Another difference on the receive side occurs in the PCL. There, the client side PCL will look at the PCL function number and dispatch to a callback (if the call was asynchronous), or pass it up to the user directly (since the watcher thread wasn't involved). On the server, only the socket watcher will get messages, and the PCL will dispatch the messages to the functions that were registered for the corresponding PCL function number. Hmm.. These are actually similar enough that maybe they could be done with the same mechanism. Then the main difference between the client and server with respect to EMS is just that the server is listening for incoming connections and uses different timeouts.

<u>Errors</u>
I can't really describe this in detail now. This will have to come out of the implementation.

In general, if an incoming message is bad in some way, it'll just get dropped. If we care, we can keep a counter that is incremented for each bad message and raise an alarm if it reaches a threshold...

<u>Multithreading/Synchronization</u>
EMS will have 2 threads internal to itself, both of which are only required when asynchronous messages are configured or EMS is being used in a server process. The first thread is the socket watcher, which is started by the EMS initialization. The second thread is the PCL dispatcher, which is started by the PCL initialization (this thread is the context in which actual requests are processed and work is done, and this could in fact be a pool of threads with a queue in front). Other layers could add their own threads if desired. All other EMS functions are fully executed in the context of the caller.
What are the shared data structures among threads? Since EMS functions could be called by different non-EMS threads, there's a good number of data structures that need to be synchronized.

First, since multiple messages may be processed at the same time, the layers above the MTL need to lock any shared data that they manipulate internally (such as IDs or nonces). Also, access to the memory buffer pools must be synchronized. Once the MTL is reached and the message is ready to be sent through the MTL, a lock is acquired (e.g. to protect the SSL_CTX structure) and only released when the bottom-level SSL or TCP interface returns, so only one thread can be blocked on a recv for a given socket at any time. The Connection's message queue and connection state must also be synchronized, as well as the socket watcher's timeout queue and active socket list. For connection-level stuff, it may be easier to lock the entire connection object (i.e. the whole time from the message body being passed to the top layer and the point where the assembled message is placed on the outgoing queue, thus covering all layers at once); this will probably be implemented first, and we can use finer grained locking if this is a performance issue.

General policies

EMS does not ensure any ordering of processing of requests. Requests of course go out in the order they were sent, but the server may process them in a different order due to thread concurrency. If ordering is desired, it needs to be implemented in a User layer via some kind of queuing mechanism.

HTTP

[mostly taken from Bhaven's HTTP document]
To make HTTP proxies happy, we'll implement a simpleminded subset of HTTP headers. For the request message, we'll use the format:

```
POST / HTTP1.1
Host: <servername>
Connection: KeepAlive
Content-Type: octet-stream
Content-length: <content_length>

<message body>
```

For the reply message, we'll use the format:

```
HTTP/1.1 200 OK
Content-Type: octet-stream
Content-Length:  <content_length>

<message body>
```

The HTTP+TCP MTL in EMS will also support a GET-type request with a URL but no message body, if the appropriate Message options are selected and the user provides the URL string, which they (or some higher layer) will have to package. The server-side HTTP+TCP MTL won't decode the URL, it'll just store it in the Message, and leave it to higher layers or the user to interpret it. This layer will assume the HTTP header will have a max size of 4kB, and report an error if it pulls more than out of the socket before reaching the body.

Also, this layer may need to emulate HTTP 1.0 behavior (i.e. no persistent connections) if the proxy isn't 1.1-friendly, but I'll leave that to be implemented later if necessary, and for now we'll notify our customers that they need a 1.1-compliant proxy server. From what we've gathered so far, most deployed proxies should be 1.1-friendly.

EStream 1.0

The current plan for eStream 1.0 is to implement the following layers:

Packaging Utility – adapt the Sun XDR routines. We'll add more for the client←→Web server interface or Web server←→SLiM interface later if required.
Procedure Call Layer – we'll just implement one, since only one is required and all communications we have planned will use it.

User Layer – unknown, but we could implement AccessToken checks at this layer, before requests are queued at dispatch time.
MTL – we will support SSL on TCP and HTTP on TCP.

Synchronous messaging could be implemented first, but pretty much all of the asynchronous messaging needs to be in place for server functionality. Therefore release 1.0 will have full asynchronous messaging support.

Data structures/Interfaces

First things first. EMS needs to be initialized on process startup.

EMSInitialize(???) – this function sets up any global stuff required.

```
class EMSBlock
{
        char * buf;
        uint32 bufsize;  // size of the alloc'd mem that buf points to, for bounds checking
        uint32 size;     // amount of buf that is occupied, i.e. pointer to where to add data.
}

class EMSMessage
{
public:
        // actual message contents
        List<EMSBlock>      MessageBuf;

        // PCL stuff for this message
        uint32 apiNum;
        uint32 funcNum;

        // MTL stuff for this message
        uint32 messageId
        (SSL related stuff perhaps)

        // HTTP stuff for this message
        boolean isRequest;
        char * URLString;

        // other stuff
        uint32  position;  //  index into MessageBuf where next layer should start reading
                           //  or writing.  Also used to indicate where more data should be
                           //  read or written to from the network if the Msg is incomplete.
        flatten(); // this function flattens MessageBuf into just one block
}
```

```
class EMSConnectionBase
{
public:
        (functions to set some of the below values)
        SendMsg(EMSMessage * msg);  // passes msg through the stack
        EMSMessage * GetMsg();
        Reset();  /* disconnects, resets all state (clears Q's), can then be reused for other
                * servers with the same MessageStack as the given Connection object */
private:
        char * destName;
        addr   destIpAddr;
        uint16  port;
        char * proxyName;
        addr   proxyIpAddr;

        boolean isClient;        // tells EMS how to behave, esp. wrt the timeout
        boolean synchronous; // actually only relevant if is a client
        uint32  timeout;         // in milliseconds
        EMSMessageStack     stack;
        EMSSocket     * sockPtr;
        EMSConnState         state;
        Queue<EMSMessagePtr>      outgoingQ;
        EMSMsgState          msgState;
        EMSMessage  *        sendingMessage;
        EMSMessage  *        receivingMessage;
}

class EMSConnection_XDR_EPC_SSLTCP  :  EMSConnectionBase
{
private:
        EMSPackagerXDR     xdr;
        EMSLayerEPC        epc;  /* for estream procedure call ? */
        EMSLayerSSLTCP     ssltcp;

        EMSConnection_XDR_EPC_SSLTCP (init stuff for XDR, EPC, SSLTCP layers
as well as dest'n name & IP address, port, timeout)
        {
                the constructor grabs the params to initialize the relevant layers, and
                places the layers in the proper order in the message stack.
        }
}

class EMSConnectionMgr
{
        [This class is used to get connection objects of a given message stack,
         by the socket watcher; actually must pass derived classes in as this type,
```

```
        i.e. the socket watcher wants a "EMSConnectionMgr", but we give it a
        EMSConnectionMgr_XDR_EPC_SSLTCP, and the right stuff happens
        via the virtual functions.]
}

class EMSPackagerBase   [just an abstract base class]
{
public:
        PutUint32(EMSBlock *, uint32) = 0;
        PutUint8(EMSBlock *, uint8) = 0;
        PutString(EMSBlock *, char *) = 0;
        [Etc., one function for every basic datatype supported.
         For custom datatype/structs, should build a routine
         for the struct that takes a packager object as an arg
         and uses its basic datatype packaging routines.]
}

class EMSPackagerXDR
{
        [actually implements the above virtual functions using the XDR spec]
}
[for outside EMS, the XDR functions will be visible as, e.g., EMSPutUint32XDR(char *
buf, int * pos, uint32), where pos points to the place in the buf to place the uint32 and it
gets incremented as appropriate, but these functions won't check bounds for buf.]

class EMSLayerBase  [another abstract base class]
{
public:
        ProcessMsgIn(EMSMessage *) = 0;  [processes an incoming message]
        ProcessMsgOut(EMSMessage *) = 0;  [processes an outgoing message]
        Reset();

private:
        int BuildHeader() = 0;
        int BodyXform() = 0;
}

class EMSLayerEPC  :  EMSLayerBase
{
public:
        EMSLayerEPC()
        {
                [starts dispatcher thread]
        }
        GlobalRegisterAPI(API #, # funcs, func table);
```

```
private:
        [table of pointers to function callback tables (one table per API),
         this table being statically allocated would mean that I'd set a max
         supported API number to be something like 256.  This table is
        static data, i.e. visible and shared by all instantiations of this class.]
}

class EMSLayerSSLTCP
{
public:
        EMSLayerSSLTCP(SSL params, TCP params)
        { ... }
private:
        uint32 curMessageID;
        SSL_CTX * context;
}

class EMSMessageStack
{
        List<EMSLayer *> layerlist;
        EMSPackager * packer;
}
[this is really just used internally by the Connection object, of little interest anywhere
else]

EMSTimeout
{
        time    deadline;
        uint32  messageID;
        EMSConnection * conn;
}

EMSSocket
{
public:
        ChangeState(EMSSocketState newState)
        [this modifies the FD_SETs below, as do the constructor & destructor.]

private:
        socket  sock;
        EMSSocketState       state;  // derived from connection state & message state
        EMSConnection *              // for other open sockets
        EMSConnectionMgr *           // for listen sockets

        static FD_SET readset;       // used by select()
        static FD_SET writeset;
```

```
        static FD_SET exceptset;
}

EMSRegisterStack(EMSConnectionMgr *, uint16 listen_port)
{
        [tells socket watcher to bind this port with the given connection queue.
        Grabs a free EMSConnection for every connection made with a client
        and associates it with the new socket]
}
```

Most of these interfaces appear because of EMS's layered structure, to support extensibility, but are not meant to be used by the EMS user, and involve lots of lower-level grungy details that are transparent above EMS. In general those coding to use EMS only need to worry about the following:

First, configure global data of layers being used, e.g.

```
EMSLayerEPC::GlobalRegisterAPI( ... )

EMSLayerSSLTCP::DoCertificateStuff( ... )
```

Then, build a connection object appropriate to the server & API you want to use, e.g.

```
foo = new EMSConnection_XDR_EPC_SSL_TCP("appserver.foo.com", &address,
4567, 1000 [timeout in ms], CLIENT, SYNCHRONOUS);
```

Then create a message object and start dumping your parameters into it:

```
msg = new EMSMessage;
msg->apiNum = EPC_API_CLIENT_APP_SERVER;
msg->funcNum = EPC_FUNC_GET_PAGE;
msg->messageId = msgid++;

EMSPutAccessToken(foo, msg, &accessToken)
EMSPutUint32(foo, msg, appId);
/* this is really an inline function that does foo->stack->packer->PutUint32(msg, val) */
EMSPutUint32(foo, msg, fileId);
EMSPutUint32(foo, msg, pageNumber);

foo->SendMsg(msg); /* msg is destroyed now! (maybe should sent &msg?) */
reply = foo->GetMsg();        /* in the synchronous case; this blocks w/ timeout*/
size = EMSGetUint32(foo, reply);
pagedata = EMSGetByteStream(foo, reply); /* this guy allocs mem for you */
```

For reuse, the above can be wrapped in its own function with the signature:

GetPageFromAppServer(foo, &accessToken, appId, fileId, pageNumber, &size, &pagedata);
When you send your next message you should reuse 'foo'. If the server had dropped the connection, it is automatically reestablished. When you're done with foo, you can Reset() it, and reinitialize it for another server, or free it.

That's generally how messages are sent, on the client or server. The server has a few extra things it needs to do at startup time:

xdr_ssl_tcp_mgr = new EMSConnectionMgr_XDR_EPC_SSLTCP;
EMSRegisterStack(xdr_epc_ssltcp_mgr, 4567);

This starts the socket watcher thread (if it doesn't already exist), and tells it to listen to socket 4567 and when a connection is made, it sets up a EMSConnection_XDR_EPC_SSLTCP to pull data off the network when it arrives. Requests coming in are handled by the appropriate layers, which were initialized globally above, so EPC will dispatch the incoming message as configured by EMSLayerEPC::GlobalRegisterAPI(...). The socket watcher will free these connections when the idle timeout expires. The only problem with this approach is that if the EMSConnection needs initialization info that is layer-specific (can't think of any right now that aren't global), there could be problems. The Connections should have default initializations that make sense so they do the right thing when messages come in (or maybe they need to have a special server mode?).

When the client uses asynchronous messaging, it must have done a EMSLayerEPC::GlobalRegisterAPI(...) as well to register its callbacks. When the first asynchronous EMSConnection is created, the socket watcher is started (if not already there). When the connection is actually established with the server, the socket watcher is informed to watch it for the reply (along with the timeout). Otherwise, there is nothing else the client needs to do (other than track message IDs to match replies and timeouts).

Testing design

Unit testing plans

The XDR packaging utility will be put together first, but little testing needs to be done directly on it, since it will be pulled right out of Sun's XDR code, with minimal adaptation. The next piece to be built will be the HTTP+TCP MTL (since this can mostly be ripped out of our prototype), and then a test client and server will be built to test it. The client will send data patterns of various sizes, and the server will just send them back. Then the eStream Procedure Call PCL will be built and tested with another test client and server at this level. Finally the SSL+TCP MTL will be constructed and tested with the null authentication/encryption/hashing configuration, which will still test basic handshaking. Once other security issues are handled, we'll enable full SSL functionality.

Beyond the basic testing, certain features need specific testing: client reply timeouts [put sleeps in server remote procedures and also just have servers not respond at all], server connection idle timeouts [put sleeps in clients], transparent connection re-establishment [just have a client reuse the connection from above], connection states [need to start making lots of requests while a connection is being asynchronously started], message states [different amounts of stuff on the queues in & out], reuse of connection objects.

Then error conditions need to be tested, since if EMS hangs, the whole system is effectively down. Evil test clients will drop connections, even get killed, send garbage data with bad headers (various possible fields), send huge messages. Of course, implementers of EPC functions need to sanity/validity check any incoming parameters of those functions. Evil test servers will respond with bad reponses as well. Coverage tools will likely come in handy, since much of EMS's code will be present to check for all kinds of errors.

Stress testing plans

Stress testing is very important for EMS. To satisfy this need, I'll need a server that implements remote procedures for the various test cases above, across more than one EPC API, and then run a bunch of the above singular clients all at once, with multiple instances of each. That will stress EMS on the server side, and should be run for many minutes (along with evil test clients) to try to expose any memory leaks and other corruption. Additionally, I'll build a client that communicates with multiple servers, basically taking the above test clients and running them in different threads in the same process, thereby stressing the single shared EMS code on the client side.

Cross-component testing plans

EMS's interaction with other components that needs to be tested besides the external interface is its interaction with proxy servers and firewalls. We'll have to implement some kind of test bed that resembles a real world setup and run EMS through it, perhaps with different potential configurations. We needs somebody familiar with proxy servers and firewalls to look at this.

Upgrading/Supportability/Deployment design

EMS will report errors to the caller as they occur (and log them if they're serious). Some errors may be fatal to the component in which case it'll have to do the right thing (do we have a common "out of memory" handler or something like that we care about?). Of course for debugging, there will be debug logging, but I'm not sure we want to put runtime checks for that in or not.

There are going to be rules and assumptions for the EMS layers that I'll have to describe so that future additional layers will conform and thus integrate properly. I'll add these to the document here once those rules are solid, which won't happen until most of the implementation is complete.

Open Issues

1. Need to figure out how to integrate turning off TCP send coalescing (or see if SSL already uses it).
2. How do shutdown of EMS system? (wait for EPC worker threads, etc?)
3. Still need to think about how newly formed connections work on the server. Likely only to fully understand once implementation is underway.
4. RealPlayer can get the proxy address from the local installed browser (so it claims), probably from the registry. Maybe we should do this too? Does this apply to SSL proxies as well?

Server Installation Requirements
Author: Bhaven Avalani
Omnishift Confidential

Pre-requisites:

1. Hardware:
   - 3 dual-processor machines with the following configurations:
     i. 400 Mhz (or higher) dual–processor CPU.
     ii. 128 MB (or higher) RAM.
     iii. 20 GB (or higher) disk space.
     iv. 10 GB (or higher) disk space on C: drive.

   (We shall call these machines A,B and C for further discussions).
2. Software:
   - Each of the machines should be loaded with Widows 2000.
   - Machine A to be configured with SQL Server 7.0.
   - Machine B to be configured with Apache(1.3.12) webserver and Tomcat(3.1) servlet engine.
   - Machines A,B and C to be configured with SQL Server(7.0) clients.
3. Software shipped by Omnishift:
   - Web Server Software:
     i. DDL script to create the EStream data model.
     ii. webserver.jar (Servlets to access the eStream database).
     iii. Sprinta2000.jar (JDBC driver from Inet software).
     iv. JSP and HTML pages for the Web server applications.
   - AppServer.exe
   - SlimServer.exe
   - Monitor.exe
   - DLLs:
     i. otdbdll.dll (Dll for ODBC database connectivity).

Installation:

1. The software provided by Omnishift will be in a zip format (for Alpha). **Unzip** the Omnishift software. This will create the following subdirectories:
   a. C:/eStream1.0/bin (All the exes and dlls will go here).
   b. C:/eStream1.0/logs
   c. C:/eStream1.0/conf (to store the flat file configurations).
   d. C:/eStream1.0/ess (to store the estream sets).
   e. C:/eStream1.0/esc (to store the estream cache).
   f. C:/eStream1.0/web (to store all web server related components).
2. Create the eStream database using the data model script supplied on Machine A.
3. Configure the eStream ODBC data source on each of the machines.
4. Configure the Apache/Tomcat server with eStream software.
5. Configure the physical machines using the Admin UI.
6. Configure the monitor services on machines A, B and C.
7. Configure the monitor setting in the database using the Admin UI.
8. Configure Slim Servers on machine A and C.

9. Configure App servers on machine B and C.
10. Deploy the eStream sets on C:/eStream1.0/ess directory.
11. Start the Slim and the App servers.

Development setup:

1. Install *Visual Café* and *Dreamweaver* from \\wkst18\download\WebGainStudio40(Trial).exe or http://www.webgain.com/.
2. Install *Tomcat 3.1* from http://jakarta.apache.org/builds/tomcat/release/v3.1/bin/.
3. Install *SQLServer 7.0*.
4. Install JDBC driver from inet software. \\wkst18\download\sprinta2000_trial.zip.
5. Install JDK 1.3 from http://java.sun.com/j2se/1.3/.
6. Set environment variable TOMCAT_HOME and JAVA_HOME to your Tomcat and JDK root directory (e.g. c:\tomcat and c:\jdk1.3).
7. Add JDBC driver path (e.g. c:\JDBCDriver\Sprinta2000.jar) and eStream application path (e.g. c:\tomcat\webapps\classes\webserver.jar) to your CLASSPATH.
8. Add eStream application into tomcat's server.xml file (e.g. add "<Context path="/estream" docBase="webapps/estream" debug="1" reloadable="true" ></Context>" with other sample applications in c:\tomcat\conf\server.xml file).
9. In Project\Option of *Visual Café*, point the deployment directory and path to your tomcat's eStream directory and file (e.g. c:\tomcat\webapps\classes and webserver.jar), and put JDBC driver path into project directories (e.g. c:\JDBCDriver\Sprinta2000.jar).

Server Installation Requirements

Author: Bhaven Avalani

Omnishift Confidential

Pre-requisites:

1. Hardware:
   - 3 dual-processor machines with the following configurations:
     i. 400 Mhz (or higher) dual–processor CPU.
     ii. 128 MB (or higher) RAM.
     iii. 20 GB (or higher) disk space.
     iv. 10 GB (or higher) disk space on C: drive.

   (We shall call these machines A,B and C for further discussions).
2. Software:
   - Each of the machines should be loaded with Widows 2000.
   - Machine A to be configured with SQL Server 7.0.
   - Machine B to be configured with Apache(1.3.12) webserver and Tomcat(3.1) servlet engine.
   - Machines A,B and C to be configured with SQL Server(7.0) clients.
3. Software shipped by Omnishift:
   - Web Server Software:
     i. DDL script to create the EStream data model.
     ii. webserver.jar (Servlets to access the eStream database).
     iii. Sprinta2000.jar (JDBC driver from Inet software).
     iv. JSP and HTML pages for the Web server applications.
   - AppServer.exe
   - SlimServer.exe
   - Monitor.exe
   - DLLs:
     i. otdbdll.dll (Dll for ODBC database connectivity).

Installation:

1. The software provided by Omnishift will be in a zip format (for Alpha). **Unzip** the Omnishift software. This will create the following subdirectories:
   a. C:/eStream1.0/bin (All the exes and dlls will go here).
   b. C:/eStream1.0/logs
   c. C:/eStream1.0/conf (to store the flat file configurations).
   d. C:/eStream1.0/ess (to store the estream sets).
   e. C:/eStream1.0/esc (to store the estream cache).
   f. C:/eStream1.0/web (to store all web server related components).
2. Create the eStream database using the data model script supplied on Machine A.
3. Configure the eStream ODBC data source on each of the machines.
4. Configure the Apache/Tomcat server with eStream software.
5. Configure the physical machines using the Admin UI.
6. Configure the monitor services on machines A, B and C.
7. Configure the monitor setting in the database using the Admin UI.
8. Configure Slim Servers on machine A and C.

9. Configure App servers on machine B and C.
10. Deploy the eStream sets on C:/eStream1.0/ess directory.
11. Start the Slim and the App servers.

Development setup:

1. Install *Visual Café* and *Dreamweaver* from \\wkst18\download\WebGainStudio40(Trial).exe or http://www.webgain.com/.
2. Install *Tomcat 3.1* from http://jakarta.apache.org/builds/tomcat/release/v3.1/bin/.
3. Install *SQLServer 7.0.*
4. Install JDBC driver from inet software. \\wkst18\download\sprinta2000_trial.zip.
5. Install JDK 1.3 from http://java.sun.com/j2se/1.3/.
6. Set environment variable TOMCAT_HOME and JAVA_HOME to your Tomcat and JDK root directory (e.g. c:\tomcat and c:\jdk1.3).
7. Add JDBC driver path (e.g. c:\JDBCDriver\Sprinta2000.jar) and eStream application path (e.g. c:\tomcat\webapps\classes\webserver.jar) to your CLASSPATH.
8. Add eStream application into tomcat's server.xml file (e.g. add "<Context path="/estream" docBase="webapps/estream" debug="1" reloadable="true" ></Context>" with other sample applications in c:\tomcat\conf\server.xml file).
9. In Project\Option of *Visual Café*, point the deployment directory and path to your tomcat's eStream directory and file (e.g. c:\tomcat\webapps\classes and webserver.jar), and put JDBC driver path into project directories (e.g. c:\JDBCDriver\Sprinta2000.jar).

eStream 1.0 Low Level Design

Software License And Management (SLiM) Server

Amit Patel
Last Modified: [redacted]
Version 2.0

Functionality

The Software License Management (SLiM) server is required to enforce licensing terms and track overall application usage. Its primary function is to grant, renew and expire access tokens, record application usage and aid in server load balancing. Its design can be broken down into three somewhat orthogonal axis:

1. Detailed specification of eStream client interfaces – the need.
2. Design and usage of <u>Server Common Services</u> (CSC) & server database –the tools of the trade. These include logging, system monitoring, thread package, encryption, TCP/IP communications, etc.
3. Core SLiM server logic that fulfils client interfaces (1) using CSC (2).

This document address items 1 and 3 in detail; item 2 should be covered in various other documents emerging from the server team.

Data type definitions

Common Data Types

- Standard atomic data types everyone (clients, builders, servers) must agree on: **Int8, Int16, Int32, Int64, uInt8, uInt16, uInt32, uInt64, uInt128** (specially for GUIDs).
  - Notice: No floating point types; I don't see a compelling reason to pass floating-point number across wire.
- **String** is represented as a size (uInt32) and it contents. Its contents must include a NULL termination character, it must be included as part of the size field.

Length	characters…..

- All eStream sets will use little endian representation.
- All complex data structures between major components (def: at least client/servers) should be version identifiable. Proposal: put a version number (uint32) as 1st word in the structure.
- Most complex structures are variable length because they contain strings. I think it would be good to put the total size (in bytes) of that structure as second word so that reader can know how much to read. If marshalling/unmarshalling utilities provide a way to represent that, it won't be needed in each struct.
- We'll need a single place where all globally (definition: across client and servers, and between servers) visible macros (#define & enum) are defined. We'll also need reserved

name spaces, and reserved number ranges for different components. Until all that is decided, I am defining macros without assigning any values.

- I am assuming that our message pack/unpack utilities will deal with alignment issues.

Server Sets

EstreamServerID contains server parameters clients need to know in order to initiate a connection. EStreamServerSet is simply a list of individual server Ids. The main use of this data structure is for SLiM server to return a list of app servers that can serve a given application. Server ids and server sets are specific to each application; client is responsible for keeping a map of app id → server set.

```
#define    SERVER_TYPE_APPLICATION
#define    SERVER_TYPE_SLIM
#define    SERVER_TYPE_ASP_WEB

typedef struct {
     uInt32    Version;
     uInt32    SizeInBytes;
     uInt32    MachineIP;
     uInt16    MachinePort;
     String    MachineName;
} eStreamServerID;

example of an eStreamServerID:
     {1,20,0x12348,80,{12,"s10.asp.com"}}

typedef struct {
     uInt32              Version,
     uInt32              SizeInBytes,
     uInt32              ServerType;
     uInt32              ServerIDCount,
     eStreamServerID     ServerID[];
} eStreamServerSet;

example of an eStreamServerSet:
{1,??, SERVER_TYPE_APPLICATION, 2,
     {1,20,0x12348,80,{12,"a10.asp.com"}},
     {1,20,0x12349,80,{12,"a11.asp.com"}}}
```

Access Tokens

This is a main data structure that is getting passed back and forth between a client and SLiM/App servers. Granting an AccessToken is an acknowledgement of client's legal right to run the application – the license. Denying an AccessToken is an acknowledging that the client does not have rights to run the application; probable causes include a user running multiple sessions, user not paying bills etc.

From client's perspective, it is totally opaque; but SLiM server uses it to pass information to the app servers so that the app server does not have to rely on the database lookups. Each access token will have a unique ID.

Terminology

Billing Granularity – Granularity at which an ASP is interested in billing its customers. Most ASPs today bill on monthly basis and eStream will assume that to be the 'norm'. However, to support things like short trial memberships, we'll design eStream to handle billing as often the AccessToken Renewal Frequency (defined below). If an end user simply purchases the eStreamed application, the billing granularity is infinite – the upper bound. EStream should not assume that billing granularity for all apps served by an ASP is the same.

AccessToken Renewal Frequency – Frequency at which the client must renew its access tokens in order to continue eStream application use. This must be tunable parameter whose upper bound is the billing granularity; it is also the smallest billing granularity we'll support. Not all access tokens are required to have the same renewal frequency.
Recommendation: **10 minutes**.
Tradeoffs:

1. This is the smallest granularity at which a client can be evicted (defined below).
2. Finer granularity may increase the number of hits to the SLiM server and adversely effect its scalability.

Eviction Notice – In general there will be times when an ASP wants to stop a user from using an eStream application, which also means stopping a user from consuming ASP server resources. Possible reasons may be:

- Lack of payment.
- Termination of a trial membership.
- To force the client into upgrading an app.
- Just because the restroom is freezing cold.

EStream infrastructure has an inherent limitation that servers can't push anything on the client. That means SLiM servers must deny an access token or its renewal, to effectively deliver an eviction notice to the client. Also, App servers may need to be informed of such evicted access tokens so that they can deny paging requests.

Decision: *After looking at some scalability numbers, we concluded that a renewal frequency of 10 minutes should not affect the overall performance and scalability of eStream system. Consequently, we don't have to communicate the list of evicted tokens to the app server since they would be invalid soon (avg 5 minutes) anyways. This simplifies server designs by reducing cross communication between slim servers and app servers.*

```
typedef struct {
  uInt32       Version;
  uInt32       SizeInBytes;
  uInt128      ATID;            // AccessToken ID - GUID
  String       UserId;          // GUID.
```

```
    uInt128      AppID;          // GUID.
    uInt64       IssueTime;      // POSIX time_t format.
    uInt64       ExpirationTime;
  } eStreamAccessToken;
```

Other Common Data Structures

In order to allow easy/automatic updates of eStream application, we need to define a protocol by which a client can be informed of app updates. This structure will also be used when installing subscribed applications on a client.

AppName, VersionName – describe the application.
Message – a short description of an application.
Flages, such as ForcedUpgrade – client must upgrade the application.
RootFileNumber - is sort of the version # of an application root directory.
RootFileMetadata - metadata of the root directory.

```
typedef struct {
    uInt32      Version;
    uInt32      SizeInBytes;
    uInt128     AppID;
    String      AppName;         // may be "Word2000"
    String      VersionName;     // may be "SP1"
    String      Message;
    uInt8       ForcedUpgrade;
    Int32       RootFileNumber;
    ????        RootFileMetadata;

} eStreamAppInfo;
```

Interface definitions

In a single process context, cross-module interfaces are easy and intuitive when defined as C/C++ procedure calls. However, for client/server (and perhaps server/server) interfaces, we need to define our own RPC-like protocol. Sameer covering this (EMS – estream messaging services) in a different design, but I want to state couple of assumptions I am making:

- Each EMS call is assigned a unique number (Int32). Codes must be uniform across all servers (i.e. no duplication of names and numbers). We should reserve some namespaces and numbers for each eStream server. Following is the current list of EMS codes between SLiM/Clients.

```
#define     EMCC_NULL
#define     EMCC_ACQUIRE_ACCESS_TOKEN
#define     EMCC_RENEW_ACCESS_TOKEN
#define     EMCC_RELEASE_ACCESS_TOKEN
#define     EMCC_REFRESH_SERVER_SET
#define     EMCC_GET_LATEST_APP_INFO
#define     EMCC_GET_SUBSCRIPTION_LIST
```

- In addition to any data (pages etc.), an EMS calls needs to return a number of codes to communicate success/errors. Following structure provides a container for returning multiple return codes. By convention, we'll put either EMCR_FAILURE or EMCR_SUCCESS in Code[0].

```
typedef struct {
        uInt32        SizeInBytes;
        uInt32        ReturnCodeCount;
        uInt32        ReturnCodes[];
} EMCReturnCodes;

#define      EMCR_SUCCESS
#define      EMCR_FAILURE
#define      EMCR_USER_AUTH_FAILED
#define      EMCR_ACCESS_TOKEN_INVALID
#define      EMCR_SUBSCRIPTION_INVALID
#define      EMCR_LICENSE_NOT_AVAILABLE
#define      EMCR_LICENSE_ALREADY_HELD

#define      EMCR_EVICTION_NOTICE
#define      EMCR_EVICTION_MUST_UPGRADE
#define      EMCR_EVICTION_END_MEMBERSHIP
#define      EMCR_EVICTION_NO_PAYMENT
```

Acquire Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_ACQUIRE_ACCESS_TOKEN	
IN	uInt128	SubscriptionID
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAccessToken	AccessToken
OUT	uInt32	RenewalFreq
OUT	eStreamServerSet	AppServerSet
OUT	eStreamAppInfo	LatestAppInfo

Client will use this interface prior to starting an eStream application to grab the license. It accepts a subscription id, which clients received when an app was subscribed, and password; it replies with at least a list of return codes and possibly, the access token, its renewal frequency and a set of servers that can serve this app.
AccessToken – Client treats them as opaque data structures and renews them within its renewal frequency.

RenewalFreq – This uInt64 is the number of seconds the access token is valid for once it receives it. You probably don't want an absolute count (i.e. # of seconds since epoch) since clients can interpret it differently due to clock skew.
ServerSet – When a client gets an access token, it will be given a list of app servers that can serve the particular app. The ServerType member of eStreamServerSet structure will be SERVER_TYPE_APP. The list is specific to each app and should be managed as such.
LatestAppInfo – SliM servers will pass information about the latest app version (root) using this structure. Refer to client eFS design for more detail. This structure will always be passed; client will ignore it if it already has the latest version.

> Note: There is a big difference between major and minor upgrades: a major upgrade would be going from word 98 to work2000 (where app ids must change) where as a minor upgrade (app ids will not change) means applying a patch or a service pack. LatestAppInfo tries to transparently propagate latter (minor upgrades) to end users without requiring end users to unsubscribe/subscribe apps. Major upgrades will require end users to go back to the ASP web server and change subscriptions. ASP can force the end user into changing subscriptions (word 98 to word 2000) using EMCR_EVICTION_MUST_UPGRADE error code.

ReturnCodes

Success:	EMCR_SUCCESS
Failure:	EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED – Can't authenticate user with specified passwd.
- EMCR_LICENSE_NOT_AVAILABLE – License is not available.
- EMCR_LICENSE_ALREADY_HELD – If the user is already holding the license, SLiM server returns this error code along with the access token that is held & its renewal frequency. Most common cause of this error is when an end user tries to run an eStream app on two different machines simultaneously. NOTE: returned token doesn't give the right to run the application and should be treated as a denial of access token. Reason for returning the token/renewal interval is to allow the client software can effectively release the token, wait some time (>= renewal frequency) and re-try.
  - The reason client has to wait is because SLiM servers will not communicate the list of 'bad' access tokens to the app server.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc. Note that no access token will be given! This may change in the future to allow some grace period.
  - EMCR_EVICTION_MUST_UPGRADE – This type of eviction means the ASP wants the end user to stop using this particular application in favor of another (major) version of it. For example, Word 98 to word 2000.

Renew Access Token

Caller:	eStream Client	
Callee:	SLiM Server	
RPC Code:	RPCC_RENEW_ACCESS_TOKEN	
IN	String	UserName
IN	String	Password

IN/OUT	eStreamAccessToken	AccessToken
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	AppServerSet
OUT	uInt32	RenewalFreq

Clients will use this interface to renew the access token before it expires. Client will specify the old access token and if there are no errors, get back EMCR_SUCCESS, a new access token, new app server set (ServerType field of eStreamServerSet structure will be SERVER_TYPE_APP) and new renewal frequency. Upon getting the new app server set, client **must** remove the old app server set for this application. If for some reason, the access token is expired, SliM server will treat this request as 'Acquire Access Token' and may return error codes possibly from that interface (this is one of the reason for asking for usernames/password).

NOTE: Unlike Acquire Access Token, it is not returning LatestAppInfo or EMCR_EVICTION_MUST_UPGRADE error codes because once the app is running, we can't upgrade apps while running.

ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID – Access token is invalid.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc.
    - NOTE: will NOT return EMCR_EVICTION_MUST_UPGRADE.
- Error codes from Acquire Access Token.

Release Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_RELEASE_ACCESS_TOKEN	
IN	eStreamAccessToken	AccessToken
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes

Client will use this interface to release the license held by the specified user. It should be called synchronously when the application exits or crashes. The reason for requiring usernames and password is to authenticate the identity of the caller against access token owner. The reason for proactively releasing tokens as opposed to just letting them expire is because releasing it allows the user to re-acquire it (on the same or different machines) without waiting for it to expire. This allows the user to do acquire -> release -> acquire without any wait.

ReturnCodes
Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID
- EMCR_USER_AUTH_FAILED

Refresh App Server Set

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_REFRESH_APP_SERVER_SET	
IN	eStreamAccessToken	AccessToken
IN	uInt8	BadQoS
IN	uInt8	NoService
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	ServerSet

App Server sets are given to a client when an access token is acquired and are automatically refreshed when an access token is renewed. However, the client can always refresh its app server sets using this interface. Potential reasons for clients to do this:

- All servers in the current server set are not responsive – NoService = TRUE
- Servers are up, but client experiences bad QoS (network delays/timouts). BadQoS = TRUE

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID

Get Subscription List

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	EMCC_GET_SUBSCRIPTION_LIST	
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	uInt32	NumberOfSubscriptions
OUT	uInt128[]	SubscriptionID[]

A client can ask for the current list of subscribed applications using this interface. SLiM server returns the number of apps subscribed and an array of subscription ids.

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED

Get Latest Application Info

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_LATEST_APP_INFO	
IN	String	UserName
IN	String	Password
IN	uInt128	SubscriptionID
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAppInfo	UpgradeInfo

Any upgrades pending? This functionality is piggy backed on 'acquire access token' interface, but there is some value in providing it as an explicit interface. SliM server will give you the latest application information block associated with the specified subscription id; the client can decide if it already has the latest root (version) or not.

ReturnCodes

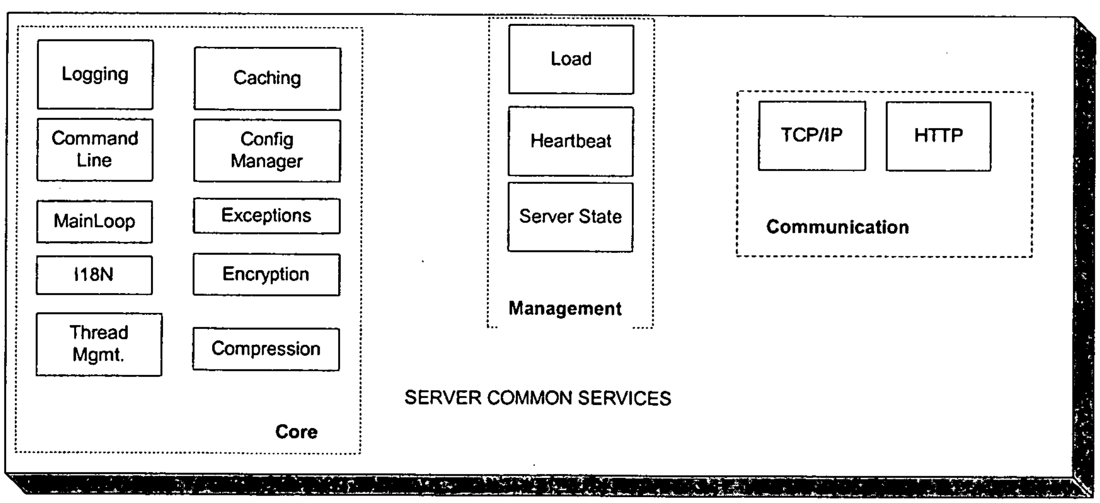
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED
- EMCR_SUBSCRIPTION_INVALID

Component design

Server Common Services

Following diagram shows the common portion of all eStream servers. Most of these boxes won't be described in this document because they are covered in specific documents.



Various Decisions

- SLiM server will uses an ODBC interface to communicate with the central database.

- From eStream client's perspective, all SLiM servers are equal in functionality. This is unlike an application server, which can be segmented to serve specific applications.
- Each SLiM server will have a unique IP/Port combination. Multiple SLiM servers running on the same machine can be distinguished by giving them different port numbers.
- SLiM servers (and app servers) will not assume any default ports; it will rely on an ASP admin to configure the port assignment. With help from OTI, ASP admin will determine how many eStream servers need to run on a machine and assign a unique number to each eStream server.

Hardware Failover

There will be a pool of SLiM servers at an ASP site; from eStream client's prospective, each of them is identical. When a client subscribes to an eStream application, it gets a set of SLiM servers to communicate with. The clients will keep this list in memory and refer to it when calling SLiM server interfaces. If it experiences difficulty communicating with a particular SliM server, it will try other servers that are part of the server set. If for some reason the server set is lost, or all servers in the set are not responding, a client can always go back to the ASP web server and refresh its server set. This gives you a transparent (from end user's prospective) hardware fail over path.

The same approach will also work for app server fail-over scenarios; specific differences are that:

1. SLiM Servers, not ASP web servers, will provide the app server set.
2. App server list will be refreshed automatically, when access tokens are renewed. This allows ASP admins to take out servers from the pool by waiting certain amount of time (>= access token renewal frequency) and not cause unnecessary client timeouts.
3. App server sets are specific to each app; SLiM servers are not.

Load Balancing

In eStream 1.0, we will not require a third party load balancers at an ASP site; we'll do minimal things at both ends (clients/server) that should be good enough for small to medium size ASPs. We may have to test with selective 3rd party load balancers to see if we can work with them or not; but this is an open issue (listed in the open issues at the end of doc.).

In eStream 1.0, we'll capitalize on the hardware fail-over mechanism to also aid load balancing. Following two actions will perform load balancing:

- When a client gets server sets (app or SLiM servers), it will distribute its hits randomly among the server in the set. In addition, clients will also get new app server sets every time they renew access tokens.
- On server side, the monitor will keep track of each server's response times to process client's requests. The data gets sorted from most responsive to least responsive and stored into the database. Top 'X' servers from this list will be given to the client when it makes an explicit request to refresh its app server set, acquires or renews an access token.

Testing design

Interface Testing

SLiM server is tightly coupled with three components: client, ODBC/Database and monitor; it is fairly difficult to isolate it from *all* of those components for unit testing. A better approach is to exhaustively test the client/SLiM server interfaces, which will in fact also test large numbers of interfaces to the other two components. The idea is to crank up a client that will make every possible SLiM server request and make sure that SLiM server responds accordingly.

I think it is good idea to create a simple testing framework (that may evolve with time) that will simulate a real client to SLiM and app servers. We can do this by writing a program that includes common (client/server) data structures definitions, links in our eStream Message Services (EMS) component and invokes various interfaces like 'Acquire Access Token'. From SLiM server's prospective, this test program is a working client.

For each client/server interface (i.e. Acquire access token) write a test case (dummy client) that will:

- Assume that we have created a dummy database that has certain users, passwords and subscriptions.
- Invoke SLiM server with all possible input permutations. This isn't too bad since most interfaces have 2 to 4 arguments.
- In the process, ensure that SLiM server returns all possible return values it can.

  For instance, lets assume that Acquire Access Token has following prototype:
  AET(uInt128 subID, String UserName, String Password);

  TEST BEGIN:
  Assert (AET(NULL, NULL, NULL) returns
  EMCR_FAILURE & EMCR_USER_AUTH_FAILED);

  Assert (AET(good_sub_id, good_user_name, good_password) returns
  EMCR_SUCCESS, an access token, its renewal freq. Etc.);

  Assert (AET(good_sub_id, good_user_name, good_password) returns
  EMCR_FAILURE & EMCR_LICENSE_ALREADY_HELD);

Stress testing plans

Stress testing in general will be common across all eStream servers. I think it will be a good idea to invest in a 3rd party tool that can simulate real-time load on eStream servers and see its responses. Rational has various tools such as Visual Test, Robot and Site Load that are worth evaluating.

Coverage testing plans

Lot of these items will apply to all eStream components and probably should be covered in a separate eStream test plan document. I am not sure if we should do these things before each component is done or wait until they are integrated. I just want to state what may be obvious so that it is documented:

- SLiM server will achieve 85% PFA coverage as measured by Rational PureCov. Tests used to measure PFA coverage will be reproducible, either by hand or via an automated test suite.
- SLiM server will resolve all memory corruption and memory leak issues as reported by Rational Purify.
- We should have test cases that will exercise all command line options for SLiM server.
- SLiM server will be code reviewed by at least two peers.

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. How do you produce GUIDs on unix servers? Should app ids, user ids, access token ids be guids or we should create them by knowing what numbers are already used?
2. Resolve big-endian – little-endian issues. Owner: Sameer
3. Meaning of 'eviction' notice is not conveyed to the end user yet. Owner: Client person – Ann.
4. Encryption impact on SLiM servers. Owners: Amit & Igor.
5. Global name space & number ranges for different components. Owner: Bhaven
6. ASCII v/s Unicode strings? Owner: Sameer.
7. Test with 3rd party load balancers to see if we work or not. Requirements for deployment team: tell us which load balancer to certify against and set them up in our future testing lab. Owner: deployment team.

eStream Web Server Load Monitoring Applet Low Level Design

Jae Jung

[redacted]

Modifie[redacted]

Functionality

One of the requirements for the eStream web server is a facility for monitoring server load (eStream requirements 3.0, 3.2, 3.4, 3.5). Per this document, this facility will be provided by a graphical load-monitoring applet that will be available for deployment at customer sites as part of the eStream web server installation.

The load-monitoring applet will present information in the following formats through a graphical interface:

- Real-time server load information plotted on strip chart
- Historical server load information plotted on line chart
- Multi-server real-time or historical load information on a line chart

Requirements

The following list details the provisional requirements for the load-monitoring applet. The remainder of this design document is based on these requirements.

1. The applet will be able to display server loads in "real-time" as load data is retrieved from the server.

2. The applet will be able to display (in a separate mode from real-time monitoring) historical load information to the extent that this information is available in the database.

3. The applet will be configurable (via applet parameters) for the following settings:
   -- Data retrieval rate, i.e., the frequency with which the applet request new load data from the web application server
   -- Chart window size, i.e., the number of datapoints shown in the chart window at any one time.

4. The applet will be capable of concurrently displaying the load of multiple servers in a clear and concise fashion.

5. The applet will support the displaying of cumulative and average load statistics for multiple servers.

6. In real-time mode, the applet will operate as a strip chart, with the fastest chart speed determined by a global configuration setting in database, typically corresponding to the frequency with which the eStream Monitor inserts load records into the database.

7. The applet will retrieve load information from the database via an http connection to the eStream web app server.

8. The applet will run in browsers with Java 1.0.x and 1.1.x support.

9. Internationalization support. Both the Applet and the backend pieces should be internationalizable.

Description

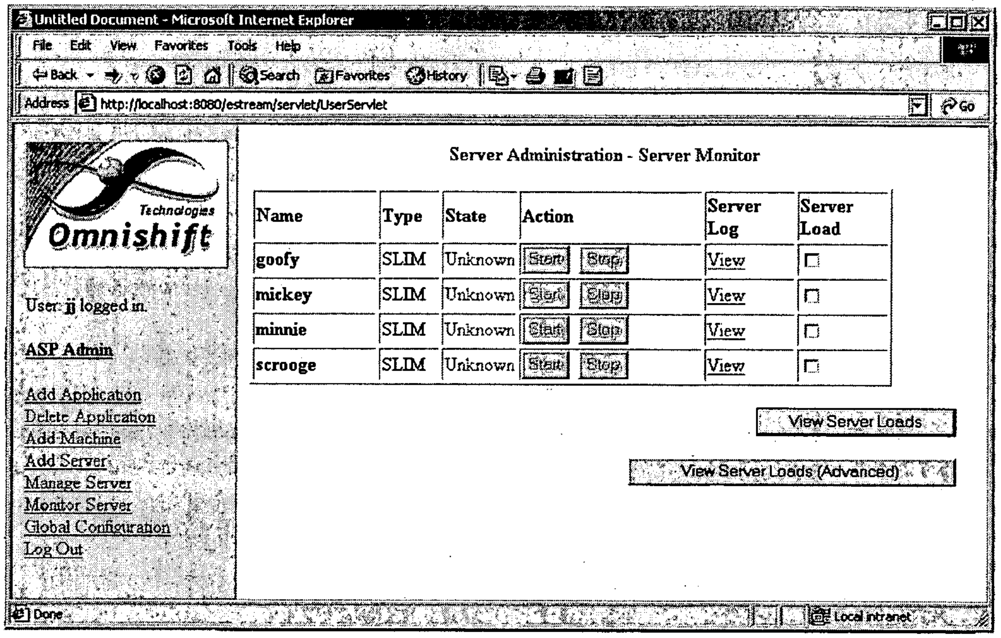
The load monitoring applet is comprised of two components. The first component will manage the retrieval of real-time or historical server load information from the eStream database. The second will consume this data and present this information to the user in a clear and concise graphical format. In addition to the applet, server-side objects will need to be written or extended to service data requests from the applet.

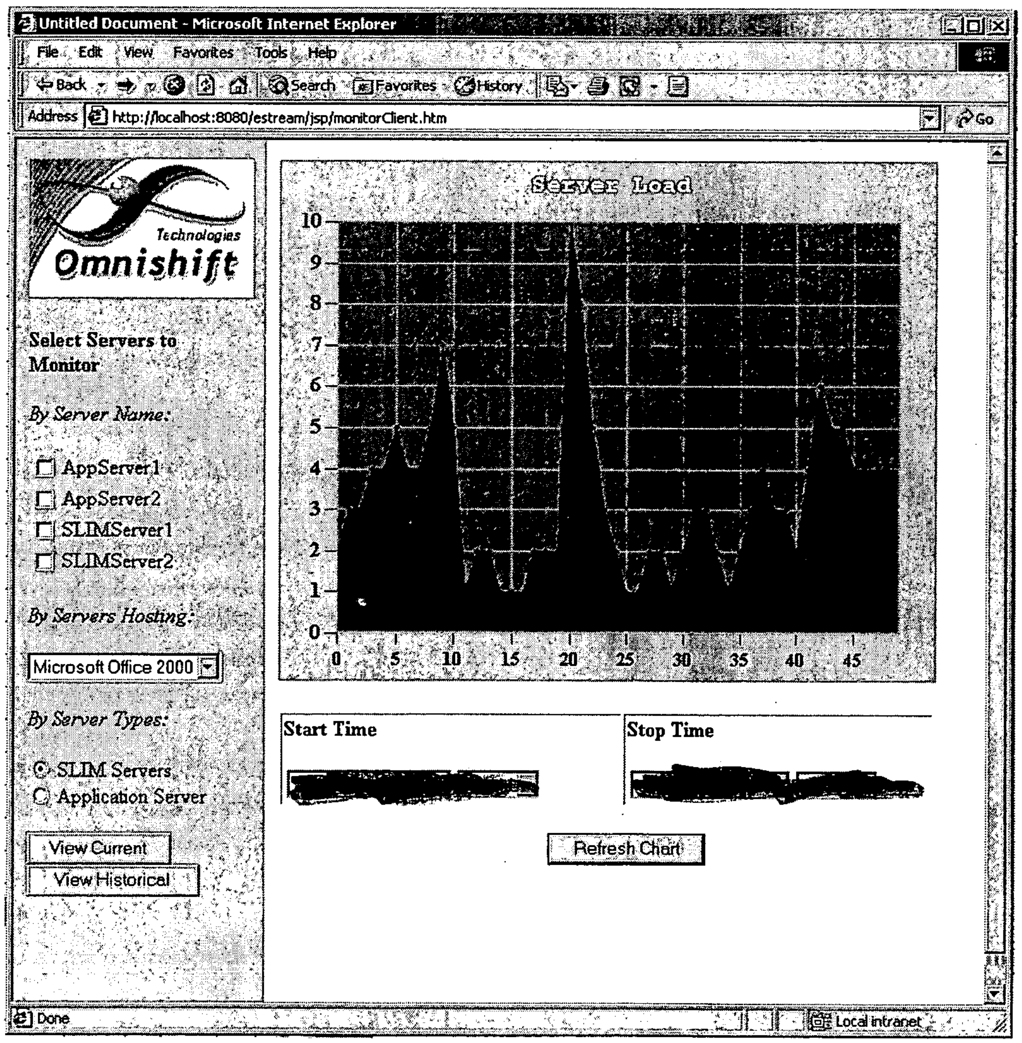
For the alpha-release of the eStream web server, the graphical presentation component will not be written internally; rather a commercially available applet or package will be used. Other aspects of the applet and the server-side components will be implemented to facilitate transitioning to an internally developed graphical component if such a decision is made for later releases.

User Interface Design

The following two screen shots are representative of the way that the load monitoring applet might be used in a particular monitoring/administration Brower interface. The first shot shows a general server administration and monitoring UI and the second shows a detailed load monitoring UI within which the load monitoring applet will be embedded.

The UI options shown in the second shot illustrate some of the reporting options for which the load monitor will be initially configurable. The applet(s) will be readily extensible to generate additional reports and more complex data combinations if desirable.


Untitled Document - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/estream/servlet/UserServlet
Go
Technologies
Omnishift
User: jj logged in.
ASP Admin
Add Application
Delete Application
Add Machine
Add Server
Manage Server
Monitor Server
Global Configuration
Log Out
Server Administration - Server Monitor
Name Type State Action Server Log Server Load
goofy SLIM Unknown Start Stop View
mickey SLIM Unknown Start Stop View
minnie SLIM Unknown Start Stop View
scrooge SLIM Unknown Start Stop View
View Server Loads
View Server Loads (Advanced)
Done
Local intranet


Untitled Document - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/estream/jsp/monitorClient.htm
Go
Technologies
Omnishift
Select Servers to Monitor
By Server Name:
AppServer1
AppServer2
SLIMServer1
SLIMServer2
By Servers Hosting:
Microsoft Office 2000
By Server Types:
SLIM Servers
Application Server
View Current
View Historical
Server Load
10 9 8 7 6 5 4 3 2 1 0
0 5 10 15 20 25 30 35 40 45
Start Time
[redacted]
Stop Time
[redacted]
Refresh Chart
Done
Local intranet

Interface Definitions

The load monitoring applet has two interfaces with other webserver components: <APPLET> tag parameters for its configuration within the web browser page and the HTTP request/response format with the web application server to request and receive load data.

1. **Applet <-> HTML Page Interface (<APPLET> Tag Parameters):**

Parameter	Req'd?	Significance	Default
hostName	Yes	Host name of webserver	None
hostPort	No	Host port of webserver	"80" (int)
chartSpeed	No	Time (in seconds) between chart scroll; also the resolution of the x-axis	"5" (int)
updatePeriod	No	Interval (in seconds) between HTTP requests for server load data. Note if chartSpeed < updatePeriod, the applet buffers load data for presentation. Maximum information latency will at most be equal to this value plus round trip time for the data request.	"10" (int)
chartWidth	No	Width (in ticks) of the applet chart; in conjunction with chartSpeed, implicitly determines the amount of data displayed	"50" (int)
mode	No	"current" for real-time monitoring and "history" for historical	"current"
startDate	No	if mode="history", the starting date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored.	none; value must be in Java Date format
stopDate	No	if mode="history", the stop date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored. Note that stopDate override the chartSpeed setting insofar as x-axis resolution is concerned.	none; value must be in Java Date format
numServers	Yes	number of servers to be plotted	1 (int, max 50)
serverName1 serverName2 ... serverName50	Yes	the id of the server(s) being plotted (up to 50 servers can be plotted at once). This **must** correspond to a existing serverID in the Server database table	none
serverData1 serverData2 ... serverData50	No	an initial set of data with which to display the initial chart. Note that these parameter(s) are the default means by which historical load data is displayed. Each set should be a comma delimited list of numbers (float)	none

- Note that applet tag parameters are all strings; where the applet does an internal type conversion, the target type format of the particular parameter has been noted.

- Note that the parameters determining the appearance of the chart (e.g., line colors, grid painting options, custom labels, etc.) are not included in this list. These parameters will either be determined by the parameters available for configuration in the commercial charting applet or TBD at some later date if the charting component is developed internally.

2. Applet <-> Web App Server Interface (HTTP)

Input:

The load monitoring applet will request load data from the webserver by executing the web server's MonitorServlet with the following parameters:

```
href="/MonitorServlet?action=getLoadData
        &serverId=...
        &startDate=...
        &stopDate=...
        &numPoints=...
```

where:

- serverId is a comma-delimited list of one or more server(s) for which load information is being requested. Note that this serverId corresponds to the serverID attribute in the Server database table.
- startDate is the (exclusive) starting date (in Java Date format) for which the load information is being requested
- stopDate is the (inclusive) end date (in Java Date format) for which the load information is being requested
- numPoints is the number of data points requested between the start and stop dates.

The applet will process the return data points as equally time-spaced points between the requested start and stop dates.

In addition, if startDate=stopDate, only one load data point (i.e., the most recent) will be returned by the servlet.

Output:

The load monitoring applet will expect load information from the web server via HTTP response in the following XML-like format:

```
<loadData>
```

```
        <serverid=a >
                <LOADLIST>
                        <LOAD value=x1/>
                        <LOAD value=x2/>
                        <LOAD value=x3/>
                . . .
                        <LOAD value=xN/>
                </LOADLIST>
        </server>
        <server id=b>
<LOADLIST>
                        <LOAD value=y1/>
                        <LOAD value=y2/>
                        <LOAD value=y3/>
                . . .
                        <LOAD value=yN/>
                </LOADLIST>
</server>
        . . .
</loadData>
```

In the above example, x1 to xN represents load values for the server a (by serverID), and y1 to yN represents load values for server b.

Testing Design

The load monitoring applet and related server-side Java code will need to be tested according to the plan outlined for other web server components in the Web Server/Database Low Level Design (WebServerDB-LLD.doc). Additionally, it should be noted that certain applet-related parameters (i.e., updatePeriod) that affect the frequency with which the applet requests load data from the web server are good candidates for tuning in order that a good balance between UI/measurement response and web server response performance be struck.

Upgrading/Supportability/Deployment Design

Upgrading/Supportability/Deployment Design will follow the model described for the Web Server in the Web Server/Database Low Level; Design (WebServerDB-LLD.doc).

Open Issues

1. How much will it cost to deploy the chosen commercial java applet/package(s) as an OEM installation? We need to find this out before we decide on a commercial

package. Also, another criteria for the choice would be: will we get support. Do we get the source code too?

2. Longer term, where's the cost/benefit breakpoint for the above where it makes more sense to write our own charting applet/package?

eStream Web Server/Database Low Level Design

Bhaven Avalani

[redacted]

Modified: [redacted]

Functionality

The eStream solution provides a set of account, user, and subscription management utilities. These utilities are provided as extensions to the ASP's(Application Service Provider) web server.

There are three categories of users for these utilities: End User, Group Administrator and ASP Administrator. The roles and the capabilities of each of these users are detailed below.

End user for a system is the user who will actually access eStream application using the eStream clients. An end user should be able to:

- Create Account and User attributes. (Username, Password, etc.)
- Change Account and User attributes.
- View all available applications in the eStream system.
- Subscribe/Manage eStream applications.
- View Account Status.
    1. List of applications subscribed.
    2. Status of current subscription.
    3. View/Change Billing information.
    4. View/Change Account information.

A *Group Administrator* is an administrator for a group of users. An individual user is by definition a group administrator for a single user group. Capabilities of a group administrator are:

- (All of single user capabilities).
- Add delete users from a group.
- Manage the active sessions for a group. A group manager should be able to release licenses from active sessions, thereby kicking out active users.
- View the billing information. This will probably need hooks to an external billing system.

An *ASP administrator* manages the overall application system. Capabilities of an ASP administrator are:

- Manage accounts/users/subscription for all users/groups in the system.
- Manage the application data for a subscription system.

1. Add new applications to the system.
2. Modify application information for the system.
3. Provide the pricing mechanism for the applications(?).

- Manage the servers in the system.
    1. Configure a server.
    2. Stop/Start a server. This is accomplished by a message to the Monitor server.
    3. Get load information for a server.
    4. Get logging information for a server.

There are essentially two different types of accounts, which the system will support: Single user account and corporate accounts.

The following licensing mechanisms will be supported by the system.

- Fixed Duration License. (Typically monthly license).
- Indefinite License.

Description

There are several key issues that need to be determined for the Web Server architecture. The options available in the market to implement these technologies are listed below.

Web Server:

- Apache
- Netscape Server
- Microsoft Internet Information Server

CGI Technology

- Servlet/JSP
    - Tomcat (from Apache group)
    - JRun (from Allaire)
- Active Server Pages (available on NT only)
- NSAPI (C level API available for Netscape and Apache).
- ISAPI (C level API available for IIS and Apache)
- CGI (Perl/C etc.).

Database Connectivity

- JDBC.
- ODBC
- Native.

Database

- SQLServer
- Oracle
- Sybase
- Informix
- LDAP(??)

The overall proposed solution for eStream 1.0 WebServer release is:

Apache + Tomcat(for JSP/Servlet) + JDBC + SQLServer.

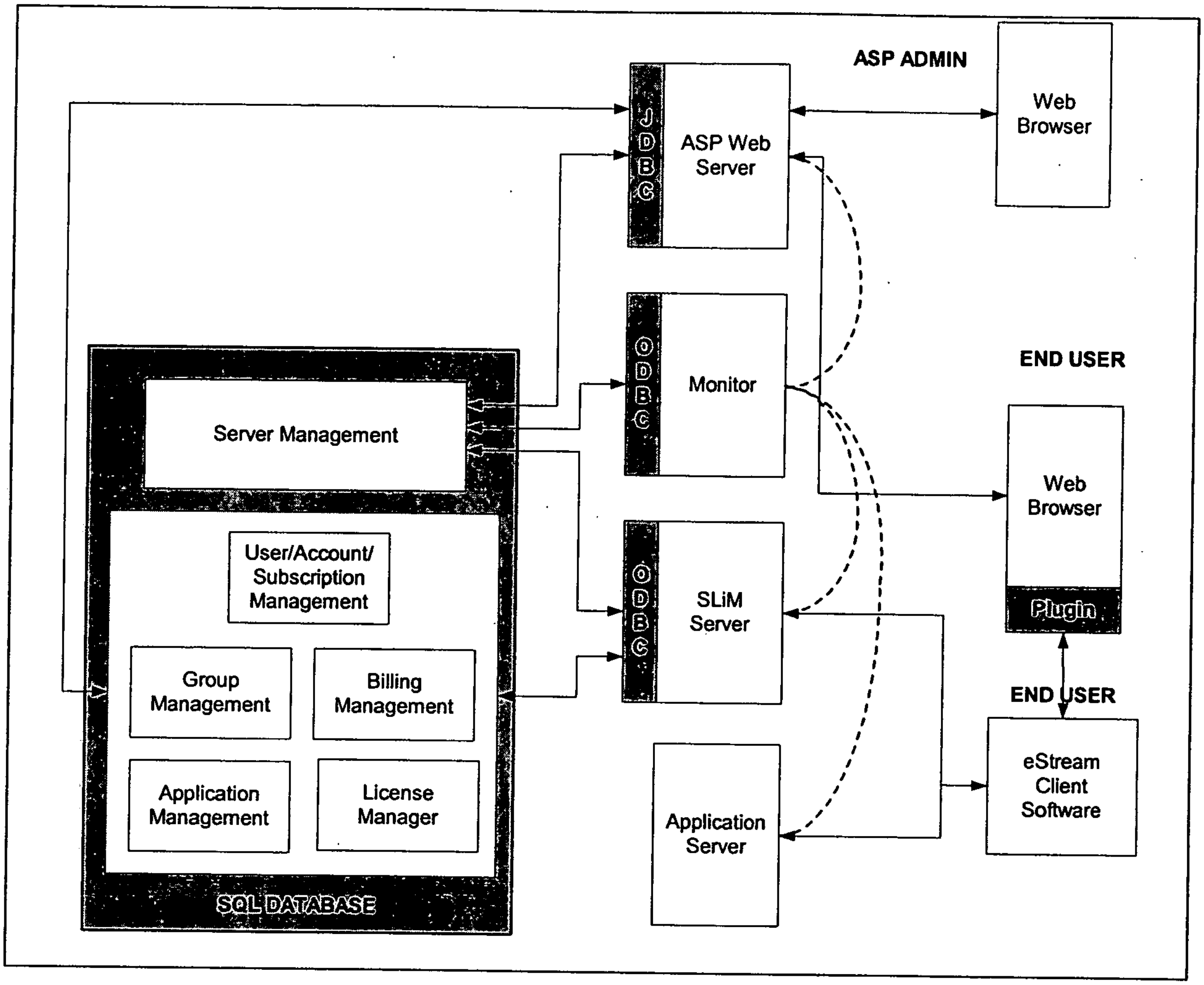
The reasons for choosing this combination for the servers are as follows:

1. JSP/Servlet is the only technology which is available for cross-platform and cross WebServer support.
2. We need to decide on a single web server to develop and test against for release 1.0. Apache is chosen to be the one as it is popular on Unix and NT platforms and it is freely available.
3. Tomcat(Apache group's reference implementation for JSP/Servlet specs) is the preferred CGI technology as it works well with Apache and all other web servers.
4. JDBC is preferred for database connectivity as its database neutral and works well with Java environment of Servlets.
5. SQLServer is the preferred database for release 1.0. This contains the scope for testing and deployment for eStream 1.0.

Since all other servers(App Server, SLM Server and Monitor) are C++ components, the following technology combination will be available for Database Access.

ODBC + SQLServer.

The data model for the eStream 1.0 database essentially consists of two high level components. The database deployment architecture is shown below:



Server Management Component: This component's primary responsibility is to manage the configuration, load and log information for a logical server in the system. The clients to this component are all the servers and administration manager. A detailed list of interfaces for this component is described in the interfaces section.

User/Account/Subscription Management: This component is responsible for maintaining the user account and subscription information for the system. The end user using the end user interface performs the updates to this component. Slim Server will access this component to validate subscriptions. A detailed list of interfaces for this component is described in the interfaces section.

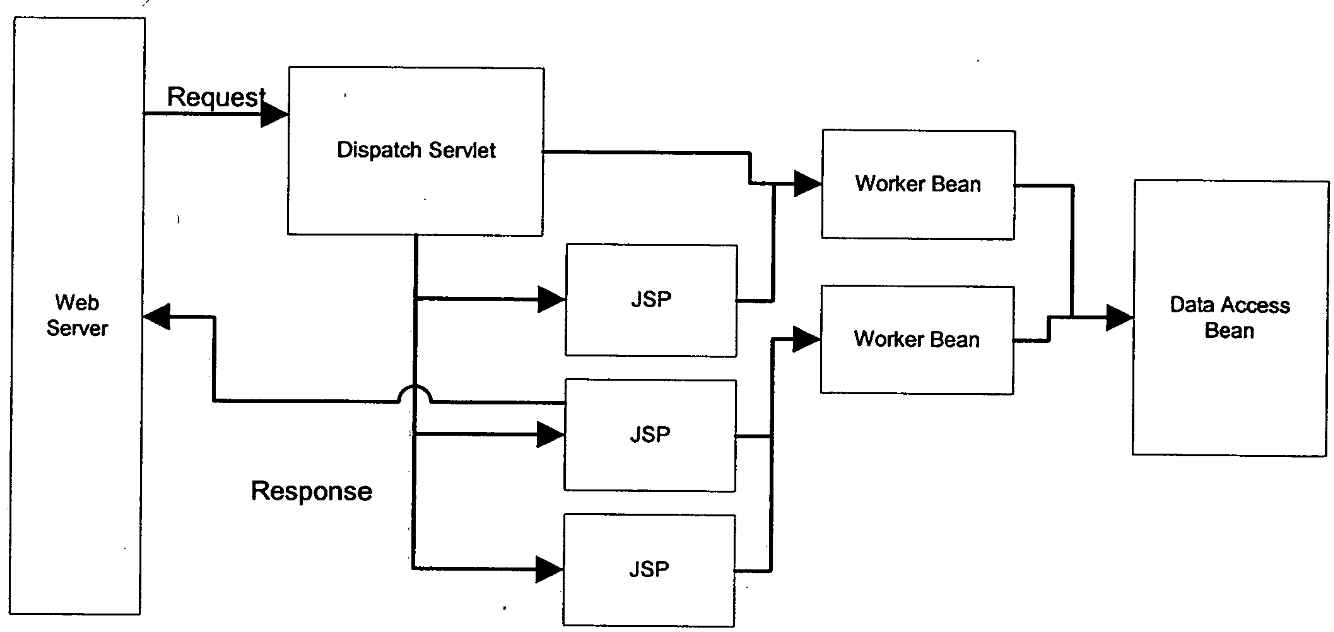
Group Management: This component is useful for managing groups of users. The group administrator can only perform updates to this component. A detailed list of interfaces for this component is described in the interfaces section.

Billing Management: This component's responsibility is to provide interfaces to an external billing system. A detailed list of interfaces for this component is described in the interfaces section.

Application Management: This component's responsibility is to provide application management interface. This component is accessible for updates only by the ASP administrator. A detailed list of interfaces for this component is described in the interfaces section.

License Manager: This component's responsibility is to manage the licenses. SLiM server will check out licenses from the license manager. A detailed list of interfaces for this component is described in the interfaces section.

The architecture for the Web Server extensions implementation is shown below:



The basic elements of this architecture are as follows:

1. Every request into the system goes through a dispatcher servlet. This servlet will perform initialization, initial validation of the request and miscellaneous checks before dispatching the request to a JSP page. A worker bean will responsible for performing the initialization. The processing of the incoming request is performed at this stage. The request is then dispatched to an appropriate JSP page.
2. The JSP page will invoke worker beans to access the dynamic data from the database via the Data Access Bean and the resultant page is sent back to the user.

This architecture is illustrated with the following example.

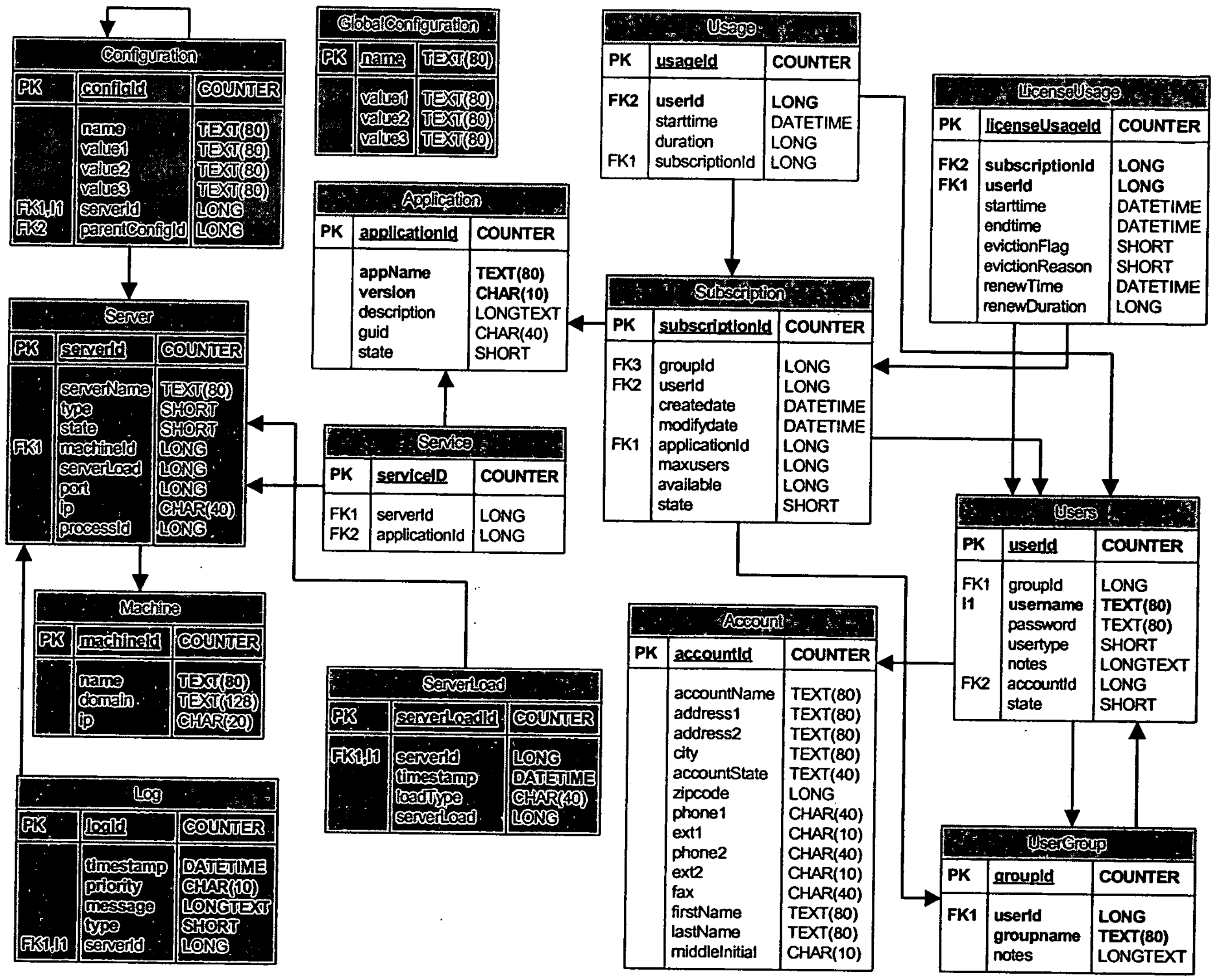
1. User sends in a request to update the username and password information in the database. Inputs are username, old password, new password.
2. The dispatch bean will call the user(worker) bean to:
   a. Validate the user's old password.
      i. The user worker bean will make a request to the data access bean to access the password for the user.
      ii. The two passwords are compared and the result is returned.
   b. If the password was valid then, update the new password.
      i. Call the data access bean to update the password in the database.
   c. Else return failure.
3. Based on the success or failure the dispatcher will dispatch the page request to the appropriate JSP page. (eg. error.jsp on failure and user.jsp on success).
4. The page will invoke the appropriate the worker bean (error bean or user bean) to obtain the dynamic data and send the response back to the user.

The salient features of this architecture are:

1. Presentation and processing logic is separate. Thus, the customer(ASP) can customize the look and the feel of the pages without impacting the processing logic as it is segregated.
2. The data access bean is separated from the worker beans, which are primarily responsible for the business logic. This allows us to change the data access layer (eg enabling LDAP access) in the future without impacting the system drastically.

Data type definitions

The central data structure for Web Server is the database model. The overall database model for user and subscription management is shown below.



The important features of this data model are:

Account: Table holding all the billing and contact information for a user or a group.

User: An end user in the system. A user can optionally belong to a group.

UserGroup: A group of users. One of the users in the group is designated as the group administrator. Each group has a unique account associated with it.

Application: This table contains the data about various applications in the supported by the ASP.

Subscription: This table contains entries for subscription items. A subscription item consists of user/group, application and license.

Usage: This table contains the runtime information for a system. SLiM server updates this table with access token usage data. A billing system may interface with this table to generate billing data. A reporting system may interface with this table to report on usage patterns.

LicenseUsage: This table is responsible for recording checked out licenses in the system.

The data model for storing the server related information is shown below:

PK: Primary Key for the table.

FK: Foreign Key. Used for relations between tables.

11,12.. : Index Columns.

Server: This table contains entries for each logical server in the system.

Machine: This table contains entries for each physical server in the system

Configuration: This table contains configuration entries for a given server. The configuration entries can be hierarchical in nature. Each configuration has the following format:

Name Value1 [Value2] [Value3] [ParentConfigId]

Load: This table maintains the historical and real-time load information for a given logical server in the system.

Service: The service table is used to record to map all the app servers and the corresponding applications that they serve.

Log: This table maintains the logs for a logical server in the system. The log messages saved here are "major" events in the logical server system. A detailed logs stored in a flat file on the physical machine containing the logical servers

Global Data Structures:

```
class DLLEXPORT Configuration
{
public:
        Cnfiguration() {mConfigId = -1; mServerId = -1; mParentConfigId = -1;};
        vrtual ~Configuration(){};
        int mConfigId;
        string mName;
```

```
        string mValue1;
        string mValue2;
        string mValue3;
        int mServerId;
        int mParentConfigId;
};

class DLLEXPORT GlobalConfiguration
{
public:
        GlobalConfiguration(){};
        virtual ~GlobalConfiguration(){};
        string mName;
        string mValue1;
        string mValue2;
        string mValue3;
};

class DLLEXPORT DBLog
{
private:
        long getTimeStamp();
public:
        DBLog();
        virtual ~DBLog();
        string mMessage;
        int mType;
        int mPriority;
        int mServerId;
        tm* mpTime; // Time stamp in ms since epoch.
};

class DLLEXPORT ServerLoad
{
public:
        ServerLoad(){mpTime=NULL; mLoadType=0; mServerLoad=0; mServerId=0;}
        virtual ~ServerLoad(){};
        tm* mpTime;
        int mLoadType;
        int mServerLoad;
        int mServerId;
};
```

The load data structure contains the following pieces of data:

- Server Id. Identifier for the server.

- Load: Response time in milliseconds.

For the Access Token and related data structures, please refer to the SLiM server Low Level Design Document. The interfaces below will discuss some of the API's based on the these data structures.

Interface definitions

The interfaces exposed by various sub-components are detailed below.

Server Management Component:

CreateServer

int CreateServer (ServerConfig* config)

```
Input:
   Server Configuration.(data structure defined
in the server configuration document.)
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
Errors:
   BAD CONFIGURATION
   NVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

UpdateServerConfig

Bool UpdateServerConfig(int serverId, String name, String value)

```
Input:
   Server Id
   Config name and value
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
Errors:
   BAD CONFIGURATION
```

```
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

AddMachine

Bool AddMachine(String name, String domain, String ip)

```
Input:
   Machine name,domain and ip.
Output:
   Success/Failure
Comments:
   Create a physical machine entry.
Errors:
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetServerLog

Bool SetServerLog(int serverId, LogTuple log)

```
Input:
   Server Id
   Log tuple(data structure in the Logging
document.)
Output:
   Success/Failure
Comments:
   Add the log data for a server
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerLog

LogTuple[] GetServerLog(int serverId, int maxrows = 25)

```
Input:
   Server Id
   Maxrows: Maximum number of rows to be re-
turned.

Output:
```

Array of Log tuples(data structure in the Logging document.)

Comments:

Get the log data for a server

Errors:

INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServers

ServerTuple[] GetServers()

Input:

Output:

Array of Server tuples(data structure defined above)

Comments:

Get all the server information

Errors:

INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

SetServerState

Bool SetServerState (int serverId, short state)

Input:

ServerId: Unique id for a server
State: State information for a server.(Defined in the server framework document.)

Output:

Bool True/False for success/failure.

Comments:

Update the database with current state information for a specified server

Errors:

INVALID SERVER ID
DB ROW LOCKED
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerState: Obtain the last known state for a specified server

short GetServerState (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   State: State information for a server.
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerConfig: Obtain configuration information for a specified server

ServerConfig* GetServerConfig (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   ServerConfig*: State information for a
server. (ServerConfig data structure is defined
in the server configuration document).
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetLoadData:

void SetLoadData (int serverId, int Load)

```
Input:
   ServerId: Unique id for a server
```

Load: Load for the server
Output:

Comments:
Monitor may call this interface to persistently store historical load data. It is still not clear if SLM and application servers will store this directly themselves.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerLoad:

void GetServerLoad (int serverId, , int maxrows = 25, int** Load)

Input:
ServerId: Unique id for a server
maxrows: Maximum number of rows to be returned.Default is 25.
Output:
Load: Load for the server
Comments:
Obtain server component load information to manage load balance.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

FlushLoadData:

void FlushLoadData (<tuples> LoadData)

Input:
LoadData tuples containing <server id, server load> values.
Output:

Comments:

```
   Used to flush aggregated load data to the
databa
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

User/Account/Subscription Management Component

CreateUser. This API is used to create user record in the system. Arguments will be Username, Password.

Bool CreateUser(String username, String password)

ValidateUser. This API is used to validate user record in the system. Arguments will be Username, Password.

Bool ValidateUser(String username, String password)

CreateAccount. This API is used to create account records in the system. Arguments will be billing address, credit card information etc.

Bool CreateAccount(int userid, <Account Information>couple[])
Input:
Username associated with the account.
An array of names and values for the account.

AddSubscription. This API is used by the end users/group administrators to subscribe to applications.

Bool AddSubscription(int userid, <Subscription Information>couple[])
Input: An array of names and values for the subscription.

UpdatePassword. Used to change user information. Password, username etc.

Bool UpdatePassword(int userid, String old-password, String new-password);

UpdateAccount. Used to update the account information. Billing Address etc.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

UpdateSubscription. Used to add additional time to a subscription.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

GetUserRecord. Used to get current user configuration.

Couple[] GetUserRecord (int userid)

GetAccountRecord. Used to get current account configuration for a user.

Couple[] GetAccountRecord(int userid)

GetSubscriptionRecords. Used to get to subscription records in a database. End user may just want to verify what they are subscribed to.

Couple[][] GetSubscriptionRecords(int userid)

Output: An array of array of couples containing the subscription information for a given user.

DeleteUser. Used to delete users who are no longer valid in the system. Typically called by the ASP admin.

Bool DeleteUser(int userid)

DeleteAccount. Used to delete un-used accounts.

Bool DeleteAccount(int accountId)

DeleteSubscription. Used by the ASP admin to remove subscriptions.

Bool DeleteSubscription(int subscriptionId)

Group Management Component

CreateGroup. This API is responsible for creating group accounts in the database. Called by the group admin user.

Bool CreateGroup(String groupName, String admin, String notes)

AddUserToGroup. Adds a user to a group.

Bool AddUserToGroup(int groupid, int userid)

DeleteUserFromGroup. Removes a user from a group.

Bool DeleteUserFromGroup(int groupid, int userid)

GetActiveSessions. Gets the active sessions for a group.

Couple[][] GetActiveSessions(int groupid)

Output: An array of array of couples containing the following information for each active session in the system

Username
LicenseId
StartTime
EndTime
Subscription

DeleteGroup(int groupid); //Deletes the group

Couple[][]GetGroupUsers(int groupid); // Gets all the users for a given group.

Licensing Component

CheckoutLicense: Checks out a license.

int CheckOutLicense(int subscriptionId, long* pStartTime, long* pStopTime)

Inputs:

SubscriptionId: Subscription id of the user.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
```

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

RefreshLicense: Refreshes a license.

int RefreshLicense(int LicenseUsageId, long* pStartTime, long* pStopTime)

Inputs:

LicenseUsageId: License usage id.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
EVICTION
```

CheckinLicense: Check in a license

Bool CheckInLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Success/Failure

Comments:

Errors:

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

ValidateLicense: Validate that the user has a license checked out.

Bool ValidateLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Yes/No.

Comments:

Errors:

INVALID USER
`INVALID SUBSCRIPTION`
`INVALID LICENSE`
`NO DATABASE CONNECTION`
`UNKNOWN SQL ERROR`

DBAcquireAccessToken

RPCReturnCodes DBAcquireAccessToken(long SubscriptionId, long* pAccessTokenId, string UserName, string Password, long* pStartTime, long* pStopTime, long* ApplicationId)

IN	SubscriptionId	Id of the subscription being used.
IN/OUT	pAccessTokenId	-1 if this is a first time access.
IN	UserName	Username string.
IN	PassWord	Encrypted Password
OUT	pStartTime	Start time for Access Token validity.
OUT	pStopTime	Stop time for Access Token validity.
IN/OUT	ApplicationId	Id of the application. –1 Default.
OUT	RPCReturnCodes	RPC Return codes.

Processing:

This is fairly complex function. The processing involved in this function call is:

- If this is the first access (ie *pAccessTokenId == -1) then **ValidateUser**
- If the ApplicationId is –1 then **GetAppId**
- If this is the first access (ie *pAccessTokenId == -1) then **CheckoutLicense**
- If this is a renewal request: **RefreshLicense**
- If there is a failure and it is due to eviction: **GetEvictionReason**

Errors:

#define	RPCR_USER_AUTH_FAILED
#define	RPCR_ACCESS_TOKEN_INVALID
#define	RPCR_ACCESS_TOKEN_EXPIRED
#define	RPCR_LICENSE_NOT_AVAILABLE
#define	RPCR_LICENSE_ALREADY_HELD
#define	RPCR_EVICTION_NOTICE

```
#define     RPCR_EVICTION_MUST_UPGRADE
#define     RPCR_EVICTION_END_MEMBERSHIP
#define     RPCR_EVICTION_NO_PAYMENT
```

DBReleaseAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Update the Usage table with the appropriate information.
- Delete the LicenseUsage record.

Notes:

- We need a mechanism to release un-released access tokens. The way to do this would be to run a stored procedure at demand and at a predefined intervals to do this cleaning up.

EvictAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Evicts an access token.

<u>Billing Component</u>

AddUsageRecord. Called by the SLM server when it releases an access token.

Bool AddUsageRecord(String username, int subscriptionId, date starttime, long duration).

GetUsageRecordsForUser. Used by external billing system.

Couple[][] GetUsageRecordsForUser(String username)

GetUsageRecordsForGroup Used by external billing system.

Couple[][]GetUsageRecordsForGroup (String groupName)

Application Management Component

AddApplication

int AddApplication(String appname, String version, String description)

Inputs:
Appaname: Application name.
Appversion: Application version
Description. Application description.
Outputs:
-1 for failure to add the application.
>0 otherwise. Application ID.
Comments:
Returns an app id for a newly added application.
Errors:
APPLICATION EXISTS
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(String appname, int version)

Inputs:
Appaname: Application name.
Appversion: Application version
Outputs:
-1 for failure to find the application.
>0 otherwise.
Comments:
Returns an app id for a given application.
Errors:
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(int SubscriptionId)

Inputs:
SubscriptionId
Outputs:
0 for failure to find the application.

>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetSubscribedApplicationIds

Int[]* GetSubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetUnsubscribedApplicationIds

Int[]* GetUnsubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids not subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetApplicationDetail

Couple[] GetApplicationDetail(int appid)

Inputs:

Application Id.

Outputs:

Array of couple for the app id containing:
{appname, appversion, description} values.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

Component design

We will discuss some complex scenarios in this section.

Subscription

Single New User

1. Create the user. **CreateUser.**
   a. If a user already exists, return error message and go back to 1.
2. Create the account for the user. **CreateAccount**
   a. Get the contact information from the user.
   b. Prompt to get the billing information. The user may decide to not give the billing information at this point.

Corporate group admin creating an account.

1. Create the admin user. **CreateUser.**
2. Create the group. **CreateGroup**
3. Create the account information for the group. **CreateAccount.**
   c. Get the contact information from the user.
   d. Prompt to get the billing information.
4. Add users to the group. **AddUserToGroup.**
   a. This method will automatically create the user if they do not already exist in the system.
   b. The list of users is accessible to the Group Admin by querying:
      i. Our database **GetUserRecords** OR
      ii. Some external database. Eg. LDAP directory.

Single User subscribing to an application

1. Validate the user. **ValidateUser**

2. Prompt to get the billing information if the billing information is not already present.
3. Get the list of un-subscribed applications. GetUnsubscribedApplications.
   a. **GetUnsubscribedApplicationIds.**
   b. For each app id returned, get the application details. **GetApplicationDetail**
4. For each additional application user wants to subscribe, call **AddSubscription**

SLiM server checking out an access token to use an application

1. Call **DBAcquireAccessToken.**

Releasing unreleased access tokens.

Unreleased access tokens will be release using a periodic stored procedure. The algorithm for stored procedure would be as follows:

1. For all records in the LicenseUsage table whose renewal time is past, call **DBReleaseLicense.** (Note that this can be a thread in SlimServer).

UI Rules

There are in general four kinds of UI components: textboxes, selects, checkboxes and radio buttons. Certain client side validations need to be applied to the entries of these components for each of the interfaces we develop in order to facilitate server development and processing. These validations are detailed as follows:

<u>Textboxes</u>

- String literal — this kind of textbox entry can only contain letters, numeric digits, underscores and spaces.
- Non-negative integer — this kind of textbox entry can only contain non-negative integers.
- IP address — this kind of textbox entry will consist of 4 textboxes, each with a char width of 3, and each of them must contain between 1 and 3 numeric digits (inclusive).
- Phone number — for the time being, we only care about US phone numbers. It will consist of 3 textboxes, the first two (char width 3) must contain 3 numeric digits each, and the last one (char width 4) must contain 4 numeric digits.
- Password — this kind of textbox entry must be at least 5 characters long.

Please note that they cannot be blank if they are required.

<u>Selects</u>

- Drop-down — at least one selection must have been made if required.
- List — at least one selection must have been made if required.
- Date — this will consist of 5 drop-down boxes, a selection must be made in each of them if a date is required, the default will be set to the current date and time on page load or reset.

<u>Checkboxes</u>

- At least one of a group must have been selected if required.

<u>Radio Buttons</u>

- One and only one of a group must have been selected if required.

Testing design

This document must have a discussion of how the component is to be tested. Some sub-sections could include:

<u>Unit testing plans</u>

The follwowing components will be unit tested:

ODBC connectivity dll for the SliM server and the Monitor. A simple C++ executable will be provided to test the SLiM server and the Monitor interfaces. The C++ executable will:

- establish connection to the database.
- simulate access token calls.
- simulate the monitor calls.

The Web servers' servlets and the JSP pages interact with the database using a set of java beans. Each of the bean will have a test interface. A simple JSP will be provided which will call all of the test interfaces for the beans. The test interface itself will be responsible to call all the interfaces that the bean provides in a predefined calling sequence.

The servlets will be unit tested using a set of html forms which will invoke the servlets.

Stress testing plans

Stress testing will be invoked using some external testing tool which can record HTTP traffic and replay the traffic for multiple users. Performix and LoadRunner are two possible choices. (There may be additional tools available).

Coverage testing plans

Coverage on the ODBC components will use the same mechanism as the rest of the server code. Pure Coverage may be used to achieve this goal.

Coverage on the Java components is an open issue. We need to investigate the appropriate tools for doing this testing.

Cross-component testing plans

The database is the central point for distribution of the data from Web server to the rest of the servers. Thus, creating the database data which can be used by Slim server and the App server will be a good source of cross component testing plan.

Upgrading/Supportability/Deployment design

We will be finally shipping just the java class files and JSP pages to the customer. It is assumed that the customer will have the appropriate web server to support JSP 1.1 and Servlet 2.2.

Open Issues

1. We have assumed that the JDBC implementation will come from Inet software. We may need to change to an alternate JDBC vendor based on pricing, quality etc.
2. Tomcat is decided to be the JSP/Servlet engine. Again, this is a freeware and may be replaced by a commercial version (Jrun from Allaire).
3. The web server will need to talk to the Monitor. The messaging component for this communication is not well defined yet.

Exhibit C

Exhibit C

Reduction to Practice

Exhibit C1

eStream Application Install Manager Low Level Design

Nicholas Ryan

[redacted]

Functionality

The Application Install Manager (AIM) is a component of the eStream client executable. It is responsible for installing and uninstalling eStream applications at the request of the License Subscription Manager (LSM). AIM uses the information contained in an AppInstallBlock to prepare the user's system for execution of a given eStream application. It creates registry entries, copies files, and updates the file spoofing database. The user can then launch his application via a local shortcut or a shortcut on the eStream drive. Uninstallation involves undoing all changes made to the user's system by AIM during installation.

Data type definitions

This component uses the AppInstallBlock, but doesn't define it. This is defined in a low-level design document for the Builder component.

The AppInstallBlock is a binary data file with a versioned interface, basically consisting of:

- a header
- a list of files to install or send to the file spoofer
- a list of registry entries to install or remove
- a set of prefetch requests to communicate to the profile/prefetch component
- a set of initial profile data to communicate to the profile/prefetch component (post-version 1.0)
- a comment section
- an embedded DLL that can be loaded and executed for custom install needs
- a section containing a license agreement to be shown to the user

Many of the AIMsc functions take an AIBFileRef as an argument, which is an opaque pointer to the following structure:

```
typedef struct
{
    HANDLE          FileHandle;
    AIBFileHeader   FileHeader;
    AIBIndexEntry   *IndexEntries;
    LPCTSTR         AppName;
```

```
} AIBFileInfo, *pAIBFileInfo;
```

It is assumed that an external header file will be available that defines structures such as AIBFileHeader and AIBIndexEntry. For now, refer to the AppInstallBlock-LLD for how they might be defined.

Also, each application has a prefetch data file created for it an install time that is initialized with prefetch data from the AppInstallBlock. This data file is named and located as described in the Component Design section, and just consists of a non-padded list of the following structures:

```
typedef struct
{
  UINT32  FileNumber;
  UINT32  BlockNumber;

} PrefetchItem, *pPrefetchItem;
```

The following data types are used in the AIM and AIMsc interfaces:

```
typedef void *AIBFileRef;
```

Error codes that are assumed to be defined somewhere are:

```
SUCCESS (0)
ERROR_BUFFER_TOO_SMALL
```

Interface definitions

Application installation/uninstallation

There are only two functions exposed by AIM, one for application installation, and another for application uninstallation. Only the License Subscription Manager will be calling these functions.

UINT32 AIMInstallApplication(UINT8 AppId[16], LPCTSTR PathToAIB)

Parameters

AppId

[in] The application ID of the eStream application to install.

PathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to install.

Return Values

SUCCESS (0) if all the actions specified in the AppInstallBlock were performed successfully, an error code otherwise.

Comments

None.

UINT32 AIMUninstallApplication(UINT8 AppId[16])

Parameters

AppId

[in] The application ID of an existing eStream application to uninstall.

Return Values

If the specified application ID is not recognized, or the original AppInstallBlock is not found, an error code will be returned. Otherwise, AIM will make an attempt to undo all of the actions it performed while installing this application. It will return SUCCESS (0) if it undid enough of these actions so that any future installation of the same application will succeed.

Comments

None.

AIM Sub-Component Interface

Much of the functionality required by the AIM design will be useful to the Builder testing framework as well. This functionality will be treated as a sub-component within the AIM component, called AIMsc, and will export a well-defined interface. That interface is defined as follows.

*UINT32 AIMscOpenAppInstallBlock(LPCTSTR PathToAIB, AIBFileRef *pAIBFile)*

Parameters

PathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to open.

pAIBFile

[out] Returns a reference to an open AppInstallBlock file.

Return Values

SUCCESS (0) if the AppInstallBlock was opened successfully and validated, an error code otherwise.

Comments

The reference returned by this function can be used as a parameter to any of the other functions that take an AIBFileRef.

UINT32 AIMscCloseAppInstallBlock(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

Return Values

SUCCESS (0) if the close succeeded, an error code otherwise.

Comments

None.

void AIMscGetAIBVersion(AIBFileRef AIBFile, UINT32 *pAIBVersion)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AibVersion field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

void AIMscGetAIBAppId(AIBFileRef AIBFile, UINT8 pAIBAppId[16])

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AppId field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

void AIMscGetAIBVersionNo(AIBFileRef AIBFile, UINT32 *pAIBVersionNo)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersionNo

[out] Returns the value of the VersionNo field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

void
AIMscGetAIBShouldReboot(
AIBFileRef AIBFile,
BOOLEAN *pAIBShouldReboot)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBShouldReboot

[out] Returns the value of the ShouldReboot flag in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

UINT32
AIMscGetAIBAppName(
AIBFileRef AIBFile,
LPTSTR pAIBAppName,
UINT16 *pSizeAIBAppName)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBAppName

[out] The value of the ApplicationName field in the AppInstallBlock is copied into the memory pointed to by this address (it will be null terminated).

pSizeAIBAppName

[in, out] On input, should point to the size of the memory at *pAIBAppName*. On output, will point to the total bytes needed to hold the entire string if ERROR_BUFFER_TOO_SMALL is returned, otherwise is undefined.

Return Values

SUCCESS (0) if the value was successfully retrieved, ERROR_BUFFER_TOO_SMALL if the buffer is too small to hold the entire string, or another error code otherwise.

Comments

None.

UINT32
AIMscCheckAIBCompatibleOS(
AIBFileRef AIBFile,
BOOLEAN *pWasOSCompatible)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pWasOSCompatible

[out] Returns TRUE if the AppInstallBlock can be installed on the current OS, FALSE otherwise.

Return Values

SUCCESS (0) if the OS version was successfully retrieved and checked, an error code otherwise.

Comments

This function will check if the currently installed operating system and service is compatible with the specified AppInstallBlock (using the compatibility information contained in the AppInstallBlock). If not, it will display a detailed message to the user and return FALSE in *pWasOSCompatible*, otherwise it will do nothing and return TRUE in *pWasOSCompatible*.

UINT32
AIMscInstallAppFiles(
    AIBFileRef   AIBFile,
    HKEY   SpoofKey,
    HKEY   SpoofRefCountKey,
    LPCTSTR   InstallLogFile,
    BOOLEAN   *pIsRebootNeeded)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

SpoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

SpoofRefCountKey

[in] An open handle to the registry key where file-spoofing reference counts are stored.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file copying, FALSE otherwise.

Return Values

SUCCESS (0) if all file install operations succeeded, an error code otherwise.

Comments

This function will perform the file copies and add the file spoofing entries specified in the File section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

For the sake of getting an error back to the user as soon as possible, this function will not undo file copies or spoof entry additions if it fails. **AIMscUninstallAppFiles** should be called to do so after the caller informs the user of the error.

UINT32
AIMscUninstallAppFiles(

AIBFileRef	***AIBFile,***
HKEY	***SpoofKey,***
HKEY	***SpoofRefCountKey,***
LPCTSTR	***InstallLogFile,***
BOOLEAN	****pIsRebootNeeded)***

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

SpoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

SpoofRefCountKey

[in] An open handle to the registry key where file-spoofing reference counts are stored.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file deletions, FALSE otherwise.

Return Values

SUCCESS (0) if enough of the file install operations were reversed so that re-installation will succeed and so that the system is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the file additions and remove the file spoof database entries specified in the install log file.

UINT32
AIMscInstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

Return Values

SUCCESS (0) if all variable modifications succeeded, an error code otherwise.

Comments

This function will perform the add/remove variable (i.e. registry entry) changes specified in the Variable section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

For the sake of getting an error back to the user as soon as possible, this function will not undo registry modifications if it fails. **AIMscUninstallAppVariables** should be called to do so after informing the user of the error.

UINT32
AIMscUninstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

Return Values

SUCCESS (0) if enough of the variable changes were reversed so that re-installation will succeed and so that the registry is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the add/remove variable (i.e. registry entry) changes specified in the install log file.

UINT32 AIMscInstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

PrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be created.

Return Values

SUCCESS (0) if prefetch block installation succeeded, an error code otherwise.

Comments

This function will install the prefetch information contained in the Prefetch section of the AppInstallBlock into *PrefetchFile*.

UINT32 AIMscUninstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

PrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be uninstalled.

Return Values

SUCCESS (0) if prefetch block uninstallation succeeded, an error code otherwise.

Comments

This function will remove the prefetch information stored at *PrefetchFile*.

UINT32 AIMscInstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

UINT32 AIMscUninstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

(NOT FUNCTIONAL IN ESTREAM 1.0)

UINT32 AIMscCallCustomInstall(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Install()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Install()*.

UINT32
AIMscCallCustomUninstall(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Uninstall()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Uninstall()*.

UINT32
AIMscEnforceLicenseAgreement(
AIBFileRef AIBFile,
BOOLEAN *pBUserAgreed)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

pBUserAgreed

[out] Returns TRUE if the user agreed to the license terms, FALSE otherwise.

Return Values

SUCCESS (0) if the license agreement was successfully displayed, an error code otherwise.

Comments

This function will extract the license agreement text included in the LicenseAgreement section of the AppInstallBlock and display it to the user. The user will be given the option to agree or not agree to the license (probably via a pair of buttons in a dialog).

UINT32
AIMscDisplayComment(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

SUCCESS (0) if the comment was successfully displayed, an error code otherwise.

Comments

This function will display to the user the comment included in the Comment section of the AppInstallBlock.

Component design

AIMsc does not have hard-coded knowledge regarding any of the standard registry and file locations used by AIM, which is why the functions in its interface take as inputs specifiers for filenames and base registry locations. Conversely, AIM itself has no knowledge of the internal structure of the AppInstallBlock file, which is why it must call AIMsc functions to work with such files.

Expansion is perform on registry entries and file paths containing certain variables, when they are read from the AppInstallBlock. These variables are defined in the Builder-LLD and will be recognized and expanded by AIM. (This includes file-spoof entries.)

AIM stores its data in the expected places for an eStream client component. All of the data AIM stores is user-specific, so it makes no use of the global locations defined for eStreams.

Registry keys

AIM stores its registry keys and values under:

HKEY_CURRENT_USER\SOFTWARE\Omnishift\eStream\AIM

This key will have its permissions modified so that ordinary users cannot modify the key (but the eStream client service will be given privileges so that it can do so). Here are the subkeys AIM places under this key:

"SpoofEntries"

Spoof entries are placed here. All spoofing is done globally, so there is no need to place it under an eStream-app specific key. Each value under this key is a pair of pathnames as follows:

<old-pathname> (REG_SZ)
- <spoofed-pathname>

"SpoofEntriesRefCounts"

Reference counting for spoof entries is done here. If multiple eStream apps are installed that want to spoof the same file, the entries must be ref-counted so that uninstall does not break the other apps. Each value under this key is a pair like this:

<old-pathname> (REG_DWORD)
- <ref-count>

Every value under SpoofEntries has a value under SpoofEntriesRefCounts with the same value name.

"<AppId>"

Every installed eStream app has its own subkey whose name is a string representation of its AppId, like so: "{00000000-0000-0000-0000-00000000000}". The values stored under each such key are:

AppId (REG_BINARY)
- AppId in binary form (16 bytes)

AppName (REG_SZ)
- name of the application (same as in the AppInstallBlock)

AppInstallBlockPath (REG_SZ)

- path to the AppInstallBlock for the application

AppInstallState (REG_DWORD)

- a value of 0 means app is installed, 1 means install is in progress, 2 mean uninstall is in progress.

Files

AIM stores per-user files at the following path:

(Path to the user's home directory)\Application Data\Omnishift\eStream\AIM

For each installed application, a separate data folder is created. The name of the folder is the AppId of the application in GUID ASCII format, like so: "{00000000-0000-0000-0000-00000000000}". The files stored under each such folder are:

<GUID string>-AIB.dat	- the AppInstallBlock file for the application
<GUID string>-Prefetch.dat	- the prefetch data file for the application
InstallLog.txt	- a generated log of what to do during uninstall

The Prefetch data file is simply an array of PrefetchItem structures (as described in the Data Structures section).

The InstallLog.txt is a list of undoable actions taken during installation. This log will be used during uninstall to determine which files and entries are safe to remove. Each line in the file contains one change, and is of the form:

ADDED or OVERWROTE or SPOOFED FILE "<filename>" (fully qualified)

ADDED or OVERWROTE KEY "<keyname>" (fully qualified)

ADDED or OVERWROTE VALUE "<valuename>" (fully qualified)

AIMInstallApplication Prototype

Installing an eStream application consists of the following steps:

1. Preparing for the installation
2. Displaying a license agreement to the user and having him agree to it
3. Installing all required local files and spoof entries for this app
4. Setting/removing registry entries as required
5. Initializing the profile and prefetch data for this app
6. Performing any required custom installation tasks
7. Displaying the comment to the user if required
8. Completing the installation
9. Rebooting the computer if necessary

AIM's policy is that if it encounters any fatal error during the execution of AIMInstallApplication, it will attempt to undo everything it did before returning. AIM also gracefully handles aborted installs and uninstalls.

Step 1 – Preparing for the installation

First, AIM checks if the application is already installed by looking for an AppId registry key for the specified AppId. If found, then the AppInstallInProgress registry value is checked. If it exists and is 1, the user is asked if he wants to re-install, otherwise, he is asked to restart an aborted or damaged installation. If the user says no, **AIMInstallApplication** cleans up and exits with an error.

Next, a free disk space check is performed to ensure that enough disk space is available for the install. The available free space must be at least twice the size of the AppInstallBlock to proceed.

Next, an AppId folder is created for the app (described earlier), and the AppInstallBlock file is copied to this folder. AIM then opens the AppInstallBlock using **AIMscOpenAppInstallBlock**. Then the AppId registry key is created and the four defined values created and initialized. The AppInstallState value in particular is set to 1 to indicate an install is in progress. If any of these operations fail, **AIMInstallApplication** cleans up and exits with an error.

Step 2 – Displaying the license

AIMscEnforceLicenseAgreement is called to display the license text to the user and ask for his agreement. If the functions fails or if the user's response is returned as FALSE, **AIMInstallApplication** cleans up and exits with an error.

Step 3 – Installing local files

The install log file to be used for this application is created or open and truncated. **AIMscInstallAppFiles** is called to copy the install files to the computer and to create the spoof entries specified in the AppInstallBlock. Handles to the spoof subkey and the spoof refcount subkey are opened and passed to this function, as well as a path to the newly created install log file. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

If it succeeds, a boolean is returned indicating whether a reboot needs to occur due to shared files being overwritten. This value is remembered for use in step 10.

Step 4 – Modifying the registry

AIMscInstallAppVariables is called to perform the registry modifications specified in the AppInstallBlock. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 5 – Initializing profile/prefetch data

AIMscInstallAppPrefetchFile is called to create and initialize the prefetch file for this application. The file has the structure specified in the Data Structures section of this document. This function takes a path to the prefetch file to be created. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 6 – Performing custom install tasks

AIMscCallCustomInstall is called to extract the custom code .dll contained in the AppInstallBlock and to call the *Install()* function it exports. If **AIMscCallCustomInstall** fails, **AIMInstallApplication** cleans up and exits with an error.

Step 7 – Displaying a comment

AIMscDisplayComment is called to display any comment to the user contained in the appropriate section of the AppInstallblock. If this function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 8 – Completing the installation

The AppInstallInProgress registry value is set to 0 to indicate the install is complete. **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and any handles to open registry keys are also closed.

Step 9 – Rebooting the computer (if necessary)

If **AIMscInstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, no reboot is performed and the application is ready to be run. **AIMInstallApplication** exits returning SUCCESS (0).

AIMUninstallApplication Prototype

Uninstalling an eStream application consists of the following steps:

1. Preparing for the uninstallation
2. Undoing all modifications done to the registry during install
3. Undoing all file copies performed during install and removing spoof entries for this app
4. Deleting the profile/prefetch data for this application
5. Performing any required custom uninstallation tasks
6. Completing the uninstallation
7. Rebooting the computer if necessary

If the uninstallation fails for any reason, **AIMUninstallApplication** will tell the user that the uninstall has failed and that he should attempt to re-install the application before trying to uninstall again.

Step 1 – Preparing for the uninstallation

First, AIM checks if the application is already installed by looking for the AppId registry key corresponding to the specified AppId. If not found, then **AIMUninstallApplication** exits with an error.

Then, the AppInstallState value is set to 2 to indicate an uninstall is in progress. **AIMscOpenAppInstallBlock** is called to open the AppInstallBlock at the path specified by the AppInstallBlockPath key. If this fails, then **AIMUninstallApplication** exits with an error.

Step 2 – Undoing registry modifications

AIMscUninstallAppVariables is called to reverse the registry modifications specified in the AppInstallBlock. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 3 – Undoing file copies and removing spoof entries

AIMscUninstallAppFiles is called to delete the files copied during install and to remove the spoof entries written then. Handles to the spoof subkey and spoof refcount subkey are passed to this function, and are where the spoof entries are removed from. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 4 – Deleting profile/prefetch data

AIMscUninstallAppPrefetchFile will be called to remove the prefetch data stored for this application. Any failure is ignored.

Step 5 – Performing custom uninstall tasks

AIMscCallCustomUninstall is called to extract the custom code .dll contained in the AppInstallBlock and call the *Uninstall()* function it exports. If **AIMscCallCustomUninstall** fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 6 – Completing the uninstallation

AIMscGetAIBShouldReboot is called and the return value saved. Then **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and the AppId folder and all its contents are deleted. The AppId registry key and all its subkeys are deleted also. Any handles to open registry keys are closed. Any failures here are ignored.

Step 7 – Rebooting the computer (if necessary)

If **AIMscUninstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, the uninstallation is complete. **AIMUninstallApplication** exits returning SUCCESS (0).

AIMsc Function Prototypes

Prototypes for the AIMsc functions declared earlier are given in this section.

UINT32
AIMscOpenAppInstallBlock(LPCTSTR PathToAIB, AIBFileRef *pAIBFile)

First, the file at *PathToAIB* is opened. Then, the header is read in, header version and size is verified, and section sizes and offsets are verified. An opaque pointer to an AIBFileInfo structure is returned in the pAIBFile parameter.

UINT32
AIMscCloseAppInstallBlock(AIBFileRef AIBFile)

The file handle at ((AIBFileInfo *) AIBFile)->FileHandle, and the AIBFile structure is freed.

void
AIMscGetAIBVersion(AIBFileRef AIBFile, UINT32 *pAIBVersion)

void
AIMscGetAIBAppId(AIBFileRef AIBFile, UINT8 pAIBAppId[16])

void
AIMscGetAIBVersionNo(AIBFileRef AIBFile, UINT32 *pAIBVersionNo)

void
AIMscGetAIBShouldReboot(
AIBFileRef AIBFile,
BOOLEAN *pAIBShouldReboot)

UINT32
AIMscGetAIBAppName(
AIBFileRef AIBFile,
LPTSTR pAIBAppName,
UINT16 *pSizeAIBAppName)

These four functions are trivial. They directly return the corresponding value of the variable in ((AIBFileInfo *) AIBFile)->AIBFileHeader. (See the interface declaration for **AIMscGetAIBAppName** for details on its calling logic.)

UINT32
AIMscCheckAIBCompatibleOS(
 AIBFileRef AIBFile,
 BOOLEAN *pWasOSCompatible)

This function will call an API such as GetVersionEx (for Windows) to determine the currently running operating system. The OS version is then converted to a bitmask (using constants defined in an external AppInstallBlock header file) and compared with the OS and Service Pack bitmaps in ((AIBFileInfo *) AIBFile)->AIBFileHeader. If the bits are present, *pWasOSCompatible* is set to TRUE, otherwise FALSE.

UINT32
AIMscInstallAppFiles(
 AIBFileRef AIBFile,
 HKEY SpoofKey,
 HKEY SpoofRefCountKey,
 LPCTSTR InstallLogFile,
 BOOLEAN *pIsRebootNeeded)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the File section. If not found, an error code is returned. Otherwise, the section is parsed.

The File section is organized as a series of trees, with directories as non-leaf nodes and plain files as leaf nodes. All nodes are stored contiguously according to the pre-order traversal of the trees.

Each directory node contains the name of a single directory and a number indicating the number of children this node has. Each file node contains the file version and file name, and a flag indicating whether the file is to be spoofed or not. If so, then the last entry in the node is the spoofed pathname, otherwise it is the actual contents of the file itself.

(The actual structure types defined for these nodes are assumed to be defined in a header file external to AIM. See the AppInstallBlock-LLD for reference.)

A directory stack algorithm will be used to parse the trees and reconstruct the directory paths. Due to the complexity of the task, several helper functions are used by the algorithm to partition this work.

For every file copied or spoof entry added by the algorithm, an entry is made to the file at *InstallLogFile*. For the sake of brevity, no mention is made of the logging in the pseudocode below.

The parsing algorithm is as follows (TOS refers to the node at the top of the stack):

```
empty out directory stack
while there are nodes to read in the File section
    read a node

    if the node is a directory
        HandleDirectoryNode(node, ...)
    else
        HandleFileNode(node, ...)

    while stack is non-empty and TOS node number of children is 0
        pop directory stack
        if stack is non-empty
            decrement number of children in TOS node
```

Here is HandleDirectoryNode(node):

```
if node directory name contains Builder/AIM defined variables
    replace variable substrings with local expansions

if directory stack is empty
    if node directory name is not fully qualified
        error
    push onto directory stack an entry with:
        -   node directory name
        -   node number of children
    else
        push onto directory stack an entry with:
        -   "TOS directory name" cat "directory name"
        -   node number of children
```

Here's HandleFileNode(node, ...):

```
if node filename contains Builder/AIM defined variables
    replace variable substrings with local expansions
```

```
if directory stack is empty
    if node filename is not fully qualified
        error
    call DoFileInstall(filename, node, ...)
else
    if number of children in TOS node is <= 0
        error
    call DoFileInstall("TOS directory name" cat "filename", node, ...)
    decrement the number of children in TOS node
```

Here's how DoFileInstall(filename, node, ...) works:

```
if the file node is a spoof entry
    if filename already exists
        if existing version is earlier
            mark for spoofing
    else  // filename does not exist
        create zero-length file at filename
        mark for spoofing

else  // file will be copied not spoofed
    if filename already exists
        if this file is a .dll
            increment .dll shared ref count in registry
        if existing version cannot be read or existing version is earlier
            mark for copy
    else  // filename does not exist
        add line to FilesLogPath file containing filename
        mark for copy

if marked for spoofing
    create spoof entry under SpoofKey
    create or update spoof refcount under SpoofRefCountKey

if marked for copy
    attempt to copy node file to client computer
    if copy fails
```

tell system to perform copy at reboot

The *pIsRebootNeeded* argument will be set to TRUE if any file copies were scheduled to happen at reboot, FALSE otherwise. Additionally, if any spoof entries were added, then an IOCTL will be sent to the spoof driver asking it to reload the spoof database.

The shared .dll reference count mentioned in the algorithm above is stored in a standard place in the Windows registry. AIM will create or increment this reference count for every non-spoofed .dll included in the AppInstallBlock (they can all be potentially shared since they will be placed outside of the eStream app directory). Each such .dll has an associated REG_DWORD value under the key at:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs

The value's name is the path to the .dll and the value's data is a integer that is the reference count for this .dll.

```
UINT32
AIMscUninstallAppFiles(
  AIBFileRef      AIBFile,
  HKEY            SpoofKey,
  HKEY            SpoofRefCountKey,
  LPCTSTR         InstallLogFile,
  BOOLEAN         *pIsRebootNeeded)
```

Currently, the *AIBFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppFiles** is simple. It iterates over the change entries contained in the log file, and undoes file copies and spoof entry additions when it is safe to do so. Here is the algorithm:

```
while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED <filename>"
        if filename is a .dll
            decrease refcount of .dll
            if refcount is 0
                mark for deletion
```

```
            else // file not a .dll
                mark for deletion

            if file is marked for deletion
                attempt to delete file
                if deletion fails
                    tell system to perform deletion at reboot

        else if the entry is of the form "OVERWROTE <filename>"
            if filename is a .dll
                decrease refcount of .dll

        else if the entry is of the form "SPOOFED <filename>"
            decrease refcount of spoof entry for this filename
            if refcount is 0
                delete the spoof entry
                if 0-byte placeholder at filename still exists
                    delete it
```

A failure to delete a file or to schedule its deletion will not cause **AIMscUninstallAppFiles** to fail.

UINT32
AIMscInstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Variable section. If not found, an error code is returned. Otherwise, the section is parsed.

The Variable section is organized as a series of trees, similar to how the File section is organized. There are two types of nodes, key nodes and value nodes. Registry keys can contain other keys and registry values, while registry values are just name/data pairs that are stored in keys. A registry value name is always a string, but its data can be stored as any one of a number of types.

Non-leaf nodes must be key nodes, while leaf nodes can either by key or value nodes. All nodes are stored contiguously according to the pre-order traversal of the trees.

(The actual structure types defined for these nodes are assumed to be contained in a header file external to AIM. See the AppInstallBlock-LLD for reference.)

A keyhandle algorithm will be used to parse the trees and create the registry keys and values. Due to the complexity of the task, several helper functions are used by the algorithm to partition this work.

For every key or value added by the algorithm, an entry is made to the file at *InstallLogFile.* For the sake of brevity, no mention is made of the logging in the pseudocode below.

The parsing algorithm is as follows (TOS refers to the node at the top of the stack):

```
empty out the key handle stack
while there are nodes to read in the Variable section
    read a node

    if the node is a key
        HandleKeyNode(node, ...)
    else
        HandleValueNode(node, ...)

    while the stack is non-empty and the TOS number of children is 0
        if TOS key handle is open
            close it
        pop the directory stack
        if the stack is non-empty
            decrement the number of children in TOS
```

Here is HandleKeyNode(node):

```
if the key name contains a Builder/AIM defined variable
    replace the variable substring with its local expansion

if the keyhandle stack is empty
    if the key name is not fully qualified
        error
    create key under HKCR, HKLM, etc. and save key handle
else
    if the number of children in TOS is <= 0
        error
```

create key under TOS key handle and save key handle

push onto the keyname stack an entry with:
- open key handle
- this number of children

Here's HandleValueNode(node, ...):

```
if the keyname stack is empty
    error
else
    if the number of children in TOS is <= 0
        error
    call DoInstallValue(TOS key handle, node, ...)
    decrement the number of children in TOS
```

Here's how DoInstallValue(key handle, value node, ...) works:

```
if the value name contains a Builder/AIM defined variable
    replace the variable substring with its local expansion
call SetValueEx(key handle, "value name", value type, value data, ...)
```

UINT32
AIMscUninstallAppVariables(
  AIBFileRef AIBFile,
  LPCTSTR InstallLogFile)

Currently, the *AIBFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppVariables** is simple. It iterates over the change entries contained in the log file, and undoes registry key and value additions when it is safe to do so. Here is the algorithm:

```
while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED <keyname>"
```

if key at keyname (fully qualified) still exists
delete it and all subkeys and values

else if the entry is of the form "ADDED <valuename>"
if value at valuename (fully qualified) still exists
delete it

Keys and values that were overwritten are not deleted, which is why those log file entries are not considered by the algorithm above.

A failure to delete one or more registry entries will not cause **AIMscUninstallAppFiles** to fail.

UINT32
AIMscInstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Prefetch section. If one is found, the prefetch data is read in and written out into the file at *PrefetchFile* as an array of PrefetchItem structures (this structure will change to match how the prefetch data items are represented in the AppInstallBlock-LLD). Any existing file at *PrefetchFile* is overwritten.

Next, the Prefetch component is called to set up an association between the new application and its prefetch file.

UINT32
AIMscUninstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

The file at *PrefetchFile* is deleted and the Prefetch component is called to remove the association between the app being uninstalled and the prefetch file.

UINT32
AIMscInstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

UINT32
AIMscUninstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

(NOT FUNCTIONAL IN ESTREAM 1.0)

UINT32
AIMscCallCustomInstall(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Code section. If one is found, the section is read in and written out again as a .dll library.

This library is loaded and the *Install()* function export is called (and its return value returned).

UINT32
AIMscCallCustomUninstall(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Code section. If one is found, the section is read in and written out again as a .dll library. This library is loaded and the *Uninstall()* function export is called (and its return value returned).

UINT32
AIMscEnforceLicenseAgreement(
AIBFileRef AIBFile,
BOOLEAN *pBUserAgreed)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the LicenseAgreement section. If one is found, the license text is read in and displayed to the user in a dialog. The user will be asked to either agree or disagree with the license, and *pBUserAgreed* will reflect his decision.

UINT32
AIMscDisplayComment(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Comment section. If one is found, the comment text is read in and displayed to the user in a dialog.

Testing design

Unit testing plans

AIM will be tested by a program that generates AppInstallBlocks designed to stress the component. AIM will be asked to install the given AIB and if successful, the resulting state of the system will be compared to the expected state had all the files and variables been installed correctly. An uninstall will then be performed and the system state also checked.

The focus of the testing will obviously be on the File and Variable sections. The other sections such as Code and Comments will be stressed also, but their boundary conditions are much simpler.

AIM's ability to gracefully handle aborted installs and uninstalls will also be tested.

Stress testing plans

The program described above can be deliberately tuned to create AppInstallBlocks of unusual size and organization. For example, AppInstallBlocks with thousands of files and registry entries, or files and entries with unusually long names, etc.

Coverage testing plans

In addition to the stress testing, deliberately malformed AppInstallBlocks will be generated by the test program to hit as much error-handling code as possible. AIM's data files and registry entries can also be deliberately mangled to help achieve this effect.

Cross-component testing plans

As soon as they are available, Builder-generated AppInstallBlocks will be tested to verify that the AIM is compatible with the Builder's output. As soon as the LSM and a browser plugin are available, the communication path from browser to LSM to AIM will be tested. As soon as the file spoofer is available, compatibility with the file spoof entries that AIM makes will be tested.

Upgrading/Supportability/Deployment design

AIM will make use of the eStream logging facility to record information about errors and other unusual conditions that occur. The log file will be useful for diagnosing problems that occur during testing and in real world situations.

If the AIM component is upgraded, it must still be able to uninstall any eStream applications installed at the time of upgrade. This entails being able to interpret old AIM registry entries and data files, including the AppInstallBlock. This is more a concern for future designers of the AIM component, however.

Open Issues

- How will the various anti-piracy strategies being considered affect the design of AIM, if at all?

Exhibit C2

eStream Client Networking Low Level Design

Dan Arai
Version1.6

[redacted]

Functionality

The Client Network Interface (CNI) provides the interfaces for sending messages to servers and provides threads for receiving responses and dispatching them appropriately. It uses the eStream Messaging Service (EMS) APIs to send and receive various messages to and from the application servers and SLiM servers.

The number of threads in the CNI will depend on the functionality available from the EMS. In particular, more threads are necessary if the EMS provides asynchronous messaging capability (and the CNI uses this interface). The interfaces presented by the CNI are identical for both cases, but the internal organization of the component is not.

The prefetcher will make calls to client networking interfaces (indirectly through ECM-ReservePage) to send requests for pages. Similarly, the LSM will make calls to acquire access tokens and subscription information.

The networking component is responsible for examining the stream of requests to it and deciding when to coalesce multiple page requests into a single request to the server.

The EMS does not provide reliability in the event of server failure. The CNI is responsible for handling server failover and reissuing failed requests on different servers. The CNI abstracts the servers from other parts of the system. Clients of the CNI don't need to specify a particular server to make a request.

Since the client networking component is where timeouts and retries occur, it is the component that controls the policies for how long we wait for a connection to time out and how many times we retry a request before giving up. These parameters will be tunable. Any other parameters of the CNI that make sense to tune will be tunable.

The CNI is also the component responsible for implementing the server selection policy.

Data type definitions

The CNI uses the request structure defined by the ECM.

The CNI maintains an internal queue of messages that must be sent to servers. This queue is not exposed outside of the CNI. Like the ECM request queue, this queue will be maintained as a circular, doubly-linked list.

```
typedef struct _NWRequest
{
     NWRequestType type;
     union {} parameters; /* params, depends on type */
     struct _NWRequest *next;
     struct _NWRequest *prev;
} NWRequest;

typedef enum
{
     CNI_PAGE_READ,
     CNI_ACQUIRE_ACCESS_TOKEN,
     CNI_GET_LATEST_APP_INFO,
     CNI_RENEW_ACCESS_TOKEN,
     CNI_RELEASE_ACCESS_TOKEN,
     CNI_REFRESH_APP_SERVER_SET,
     CNI_GET_SUBSCRIPTION_LIST
} NWRequestType;
```

The CNI provides an enumeration of the parameters that can be tuned. This enumeration is expected to grow as the number of tunable parameters grows.

```
typedef enum
{
     CNI_NUM_RETRIES,
     CNI_TIMEOUT,
     CNI_PROXY_ADDRESS,
     CNI_EFFECTIVE_BANDWIDTH
} NWTunableParameter;
```

Related Components

The prefetcher and LSM call on the CNI to send requests to the app and SLiM servers. The CNI makes calls to the ECM and LSM to inform them of responses that have come back from the server. The CNI will also make calls to EFSD interface functions when pages come back that satisfy EFSD requests.

Interface definitions

CNIGetPage

eStreamStatus CNIGetPage(
  IN ApplicationID *app*,
  IN EStreamPageNumber *page*
);

CNIGetPage is the interface used by the ECM function **ECMReservePage** to request that a page be sent by the server. (**ECMReservePage** is called indirectly by the pre-

fetcher.) Note that no distinction is made between prefetches and demand fetches. To prevent race conditions or deadlock, the requested pages must already be marked as "in flight" in the index, and any requests for these pages from the EFSD must already be on the "in flight" queue before calling this interface.

The CNI is responsible for selecting a server to direct this request to, and resending in the event of network or server failure. It will coalesce requests for multiple pages from the same application into a single request to the server.

CNIGetSubscriptionList

eStreamStatus CNIGetSubscriptionList(
    IN string *Username,*
    IN string *Password*
);

CNIGetSubscriptionList enqueues a request to acquire a subscription list from a SLiM server. When the subscription list is returned by the server, the client response thread will notify the LSM of the returned data via a callback defined in the LSM document.

CNIGetLatestApplicationInfo

eStreamStatus CNIGetLatestApplicationInfo(
    IN uint128 *SubscriptionID*
);

CNIGetLatestApplication enqueues a request to get the latest application information for a particular app. When the server returns the result, the CNI will notify the LSM of the returned data via a callback defined in the LSM document.

CNIAcquireAccessToken

eStreamStatus CNIAcquireAccessToken(
    IN uint128 *SubscriptionID,*
    IN string *Username,*
    IN string *Password*
);

CNIAcquireAccessToken will cause the CNI to contact a SLiM server to retrieve an access token. The CNI is responsible for issuing retries if no response is received for a request. The CNI will call the appropriate LSM callback function when the data come back.

CNIRenewAccessToken

eStreamStatus CNIRenewAccessToken(
    IN AccessToken *Token,*
    IN string *Username,*
    IN string *Password*
);

CNIRenewAccessToken will enqueue a request for access token renewal. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIReleaseAccessToken

eStreamStatus CNIReleaseAccessToken(
IN AccessToken *Token*,
IN string *Username,*
IN string *Password*
);

CNIReleaseAccessToken will enqueue a request for releasing an access token. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIRefreshAppServerSet

eStreamStatus CNIRefreshAppServer(
IN AccessToken *Token*,
IN uint32 *BadQOS,*
IN uint32 *NoService*
);

CNIRefreshAppServerSet will enqueue a request for refreshing the app server set. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

The client networking component will also have routines for getting and setting tunable parameters.

CNISetParameter

eStreamStatus CNISetParameter(
IN NWTunableParameter *type*,
IN void **value*
);

CNISetParameter sets a parameter. The actual type of *value* is determined by *type*.

CNIGetParameter

eStreamStatus CNIGetParameter(
IN NWTunableParameter *type*,
OUT void **value*
);

CNIGetParameter queries the current value of a parameter. The actual type of *value* is determined by *type*.

Component design

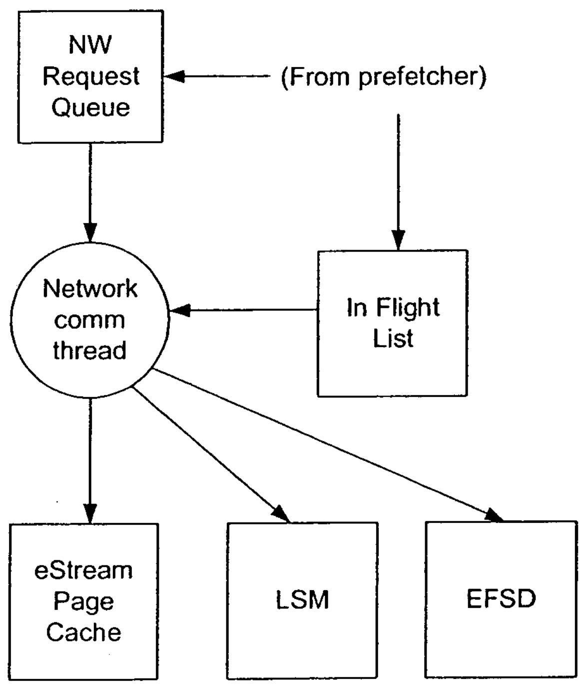
The internal organization of the client networking depends on the mechanisms available from EMS. Internally, the CNI interface functions put requests on a queue, and one or more threads services these requests by using the EMS to send messages to servers.

Synchronous Server Calls

If EMS only provides a synchronous messaging service, a single thread will be used to perform all necessary actions. The CNI interfaces will put appropriate requests on the network request queue. They will also wake up the network communication thread, if necessary.

The network communication thread's job is relatively simple. When it wakes up, it performs the following tasks:
- choose a set of requests to be coalesced and remove these from the request queue
- retrieve a server set via LSMGetAppServerSet or LSMGetSLiMServerSet, and choose a particular server for this request
- make a synchronous EMS call to send the request
- dispatch the response to the appropriate LSM or ECM callback

If the synchronous messaging mechanism becomes a performance bottleneck, we can have multiple network communication threads to increase concurrency.



Asynchronous Server Calls

The asynchronous case is a little bit more complex. Because of the proposed asynchronous call architecture, the client NW requires three threads. The CNI interfaces work just as they do in the synchronous case. They put requests on the network request queue, and wake up the network send thread. However, the actions performed by the CNI's worker threads differ in the asynchronous model.